INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Technical Challenges of Microservices Migration

JOÃO CARLOS RIBEIRO DIAS NEVES
Outubro de 2019

POLITÉCNICO
DO PORTO

Technical Challenges of Microservices Migration

João Carlos Ribeiro Dias Neves

Dissertation to obtain the Master's degree in Informatics Engineering, Specialization in Software Engineering

Supervisor: Isabel Azevedo

Porto, 2019

In dedication to all the human beings that fight for what they believe in, exchanging excuses and complaints by proactivity, hard work and determination. Also dedicated to coffee and loud music.

Abstract

The microservices architecture is a recent trend in the software engineering community, with the number of research articles in the field increasing, and more companies adopting the architectural style every year. However, the migration of a monolith to the microservices architecture is an error-prone process with a lack of guidelines for its execution. Also, microservices introduce a lot of different challenges that are not faced when following a monolithic architecture.

This work aims to fill some gaps in current microservices research by providing a catalogue of the currently most common challenges of adopting this architectural style, and possible solutions for them. For this reason, a systematic mapping study was executed analysing 54 different articles. Also, 30 industry professionals participated in a questionnaire regarding the topic. Furthermore, a participant observation experiment was performed to retrieve additional industry data.

Moreover, one of the identified challenges – distributed transactions management – was further detailed and a solution implemented using the choreographed saga pattern. The solution is publicly available as an open-source project.

Finally, multiple experts in the microservices field validated the results of the research and the distributed transactions solution and provided insights regarding the value of this work.

Keywords: microservices, migration, distributed transactions, saga pattern, systematic mapping study, industry survey

Resumo

A arquitetura de microserviços é uma tendência recente na comunidade de engenharia de software, com o número de artigos publicados sobre o tema a aumentar, assim como o número de empresas a adoptar o estilo arquitetural todos os anos. No entanto, o processo de migração de um monolito para uma arquitetura orientada a microserviços tem um alto potencial de erros, uma vez que existe falta de orientações sobre como conduzir o processo corretamente. Para além disso, os microserviços introduzem muitos desafios diferentes que não são enfrentados no desenvolvimento de um sistema monolitico.

Este trabalho pretende preencher algumas destas lacunas na investigação da arquitetura de microserviços através da construção de um catalogo dos principais desafios enfrentados ao adoptar o estilo arquitetural e soluções possíveis para estes. Por este motivo, um *systematic mapping study* foi desenvolvido, analisando 54 artigos diferentes. Para além disso, 30 profissionais da industria responderam a questionario sobre o tema. Finalmente, para obter dados adicionais da indústria, uma experiência de migração foi realizada e observada de forma ativa.

Ainda, um dos desafios identificados – gestão de transações distribuídas – foi detalhado e uma solução implementada usando o padrão de sagas coreografadas. A solução está publicamente disponível como um projecto *open-source.*

Finalmente, vários peritos em microserviços avaliaram os resultados deste trabalho, incluindo a solução desenvolvida para gestão de transações distribuídas, e deram *feedback* relativamente ao valor deste trabalho.

Palavras-chave: microservices, migração, transações distribuidas, saga pattern, systematic mapping study, questionario à industria.

Acknowledgements

First of all, I must thank with the deepest love to my mother and father, who showed me the value of hard work for the things and people we love and always managed to provide everything I ever needed to achieve all my objectives while improving as a human being.

I would also like to acknowledge my big sister, who has always been my second mother, for showing me the value of logical reasoning and helping me improve it since I was a baby.

Also, I would like to thank my life partner for all the support, patience, and continuous encouragement that she provided through the process of researching and writing this thesis. I apologise for all the hours that I invested in this work and not on her, and for all the deep breaths heard.

Furthermore, I must thank my thesis advisor Isabel Azevedo of ISEP, who instantly answers e-mails with all the help anyone would ever need and is always ready for a meeting to discuss the work. Without her availability to help when I ran into trouble, this accomplishment would probably not be achieved.

Moreover, I want to thank all the amazing human beings and extraordinary professionals that have taught me all I know, during all the years of study and work, which allowed me to become the Software Engineer I am today.

I would also like to thank all the participants of the industry survey for providing me with valuable data who made this work possible.

Finally, I must thank the experts who were involved in the validation survey for this project. Without their passionate participation and input, the validation survey could not have been successfully conducted.

Thanks to all of you, who made this accomplishment possible.

Table of Contents

Table of Figures

Table of Tables

Acronyms and Glossary

Acronyms

ACID Atomicity, Consistency, Isolation, and Durability. An ACID transaction consists of a group of requests in which all of them must be successful. This mechanism ensures database consistency by coordinating multiple requests. If one fails, all the previous ones are rollback. For this reason, either all of the requests are successful or all fail, and the database remains consistent (Richards, 2015).

CRM Customer-relationship management.

DevOps DevOps constitutes a methodology focused on unifying software development (Dev) and IT Operations (Ops) (Trihinas et al., 2018). It is composed of a set of practices that use automation and monitoring to improve the efficiency of the software creation process (Trihinas et al., 2018).

ERP Enterprise resource planning.

ESB Enterprise Service Bus.

SI Sample Issue.

SOA Service Oriented Architecture.

Glossary

Organizational Agility *"capacity to flexibly respond to changes in the environment by quickly adjusting product and service offerings"* (Baškarada et al., 2018)

1 Motivation

This chapter has the objective of introducing the work described in this document. It contains the motivation context and the structure of this document.

1.1 Context

Over the years, there has been some debate on the comparison of monolithic application architecture with modular application architecture and the advantages and disadvantages of each one (Strimbei et al., 2015).

Regarding monolithic architecture, the literature does not define the term accurately. In 1998, Aoyama referred to monolithic architecture as the "conventional" approach (Aoyama, 1998). He argued that systems should be developed towards a more "component-based software engineering" (CBSE) approach to take better advantage of object-oriented software reuse possibilities. He also lists this architectural style along with the waterfall software development approach, stating that this was the old style of software development - before the rise of the internet technology - which goes against the needs of the market with the widespread usage of internet and personal computers (Aoyama, 1998). However, Aoyama never clearly defined the monolithic architecture. In 2012, having as a starting point the Aoyama article, Lake elaborates this definition, considering that a monolithic system consists of an integrated architecture where all the fundamental application elements are organised together in a single executable or unit (Lake, 2012).

In his vision, in a monolithic system *"the user interface elements can be mixed with the program logic, and the data management code"*, he argues that this approach has the advantage of having lower complexity on the interaction between the different modules of the system as they are gathered in a single unit of software (Lake, 2012). Also, it is easier to understand a specific process as all the elements of it can be found in the same codebase (Lake, 2012). In 2017, the monolith was more formally defined: *"A monolith is a software application whose modules cannot be executed independently."* (Dragoni et al., 2017).

In general, the “monolith” and “monolithic” terms are used for lack of a better designation to refer to a system in which the different architectural elements are together in a single executable, unit or block (Strimbei et al., 2015).

Due to this nature, these system components are maintained and packaged together, distributed and deployed as a whole (Dragoni et al., 2017). Some of the benefits of this architectural style were mentioned before. However, this approach also suffers from different issues, for example:

- Any change in any module of the application requires the entire system to reboot, which can cause downtimes (Dragoni et al., 2017).
- Usually, the strategy to increase the capacity of an application to handle more requests simultaneously is to create more instances of the same software, splitting the load between them. When a performance bottleneck is detected, it is usually produced by only one of the modules. However, with a monolithic approach, the entire system must be replicated instead of a single module, which is naturally a waste of resources (Ren et al., 2018).
- Monolithic applications size grows over the years, which can cause the system maintainability to be reduced as its complexity increases if good software design practices are not followed. It leads to a *“product unmaintainable with a reasonable effort”* (Fritzsch et al., 2018).

The increasing use of cloud computing environments and hardware virtualisation makes this issues more relevant, as the industry moves their efforts of software development towards better scalability and reduced infrastructure costs (Dragoni et al., 2017).

In order to create competitive advantage, it is becoming increasingly critical for companies to have that kind of flexibility on their systems, in order to achieve greater organisational agility – *“capacity to flexibly respond to changes in the environment by quickly adjusting product and service offerings”* (Baškarada et al., 2018).

For all these reasons, some systems get to a certain point where there is an identified need for restructuring this kind of systems by researchers of the field (Strimbei et al., 2015).

Microservices oriented architecture has been regarded as a promising solution (Baškarada et al., 2018) that conjugates scalability, maintainability, ease of deployment, reduced infrastructure costs, technology heterogeneity, resilience, reusability, among others (Carrasco et al., 2018).

Some authors still suggest that software development should begin with a monolith, but over time and with better knowledge of the system complexities it should be migrated to a Microservices oriented architecture to avoid the limitations of a monolithic architecture (Fowler, 2015a). This pattern is usually called “Monolith First”.

The microservices architectural style is a recent trend in the software engineering community since it was first publicly proposed by Fowler and Lewis in 2014 (Fritzsch et al., 2018). However,

the term was first discussed at a workshop near Venice in 2011 where different authors and experts of the field debated some techniques similar to the microservices architecture, including some of the SOA principles (Fowler and Lewis, 2014). These techniques and principles were consolidated in the microservices architectural style and that definition started the referred trend.

The literature defines microservice as a small application (generally less than a couple of thousand lines of code) with a single responsibility (a functional, non-functional, or cross-functional requirement) that can be independently deployed, scaled, and tested (Baškarada et al., 2018). It must be cohesive and independent of other processes, interacting with them via messages using a clearly defined interface (Dragoni et al., 2017).
In 2017 a microservice architecture was defined as "a distributed application where all its modules are microservices" (Dragoni et al., 2017).

On the microservices architectural style, each module of the system must be identified and isolated on a single microservice. Therefore, the functionality must be divided through the services using the appropriate granularity, to achieve high cohesion inwards and loose coupling outwards (Fritzsch et al., 2018).

Even though a microservice is usually not as complex as a monolithic system, the microservice does not constitute a system by itself, so this comparison is misleading (Baškarada et al., 2018). The complete microservice architecture is composed of all the microservices communicating between each other, and that is what constitutes the system (Dragoni et al., 2017). Therefore, the development of a microservice is more straightforward than the construction of a monolithic system, but it must be deployed and integrated with the rest of the system to have value (Carrasco et al., 2018).

Following this architectural style, the following advantages can be achieved (Ren et al., 2018):

- All the components of a system are deployed independently, and each one can follow a different technology stack
- When a bottleneck is identified on a single component, the available resources can be used to replicate it, which was not possible with the monolithic approach, and naturally leads to better usage of the available resources.

Furthermore, the flexibility that microservices allow contributes to better reusability and makes it easier to replace a single component of the system, without necessarily having downtime, like on monoliths (Carrasco et al., 2018). The authors highlight the possibility of the development team to use different technology stacks on each component.

For all these reasons, different companies are migrating their monolithic systems to this microservice architecture, including Amazon, Netflix, Google, IBM, Uber, Alibaba, among others (Ren et al., 2018).

1.2 Document structure

This document is divided into 9 different chapters, which are followed by references and appendix sections.

1. This first chapter introduces the reader to the developed work by presenting the motivation context and the document structure.
2. Value analysis - presents the value analysis of this work, containing the different steps of the new concept development model of Peter Koen and a business model canvas of the project.
3. Background- describes different crucial concepts related to this work that may help the readers understand the following chapters.
4. State of the art – the most recent stage in microservices adoption research and distributed transactions management solutions are described and compared, along with related technologies.
5. Problem statement – describes the problem to be addressed, the work objectives and contributions, and the work methodology used.
6. Microservices migration research - the design of the performed research is defined and justified. Also, the systematic mapping study, industry questionnaire and participant observation study are described, and the results analysed.
7. Distributed transactions solution – the analysis of the implemented solution is described along with a technical description of the developed software.
8. Evaluation - evaluates the quality of the final work using an industry questionnaire answered by experienced professionals of the field and hypothesis testing using the questionnaire provided data.
9. Conclusions – describes the conclusions obtained with the outputs of this work. In this chapter, the achieved objectives are described along with the difficulties faced during this project, contributions of the accomplished work and future work that can be done or continued in this topic.

2 Value analysis

This chapter describes the value analysis of this work. “Value analysis is a systematic, formal and organised process of analysis and evaluation” (Rich and Holweg, 2000) of possible solutions to a specific problem with the purpose of improving the value of a product. Therefore, this chapter has the objective of analysing the value for the customer this work creates.

In order to create value, this analysis verifies if the product meets the needs of the customer and increases the product value by reducing the costs and/or improving product performance. Reducing costs that bring no benefit to the customer and that do not have any impact on the product performance naturally increase the profit and therefore the value provided by the product.

On the following sections, the value analysis will be supported by the use of the New Concept Development (NCD) model of Peter Koen (Koen et al., 2001). Furthermore, the value proposition will also be described and illustrated by a business model canvas.

2.1 New Concept Development Model

The NCD model was developed to define best practices in the innovation process of creating or establishing a product. This model provides a method to improve this process by defining a universal language that distinguishes the different stages of an iterative process of innovation (Koen et al., 2001).

The model consists of three key components:

- Five controllable key activity elements
    - Opportunity identification;
    - Opportunity analysis;
    - Idea generation and enrichment;
    - Idea selection;

    - Concept definition
- The engine that powers the elements (leadership, culture, and business strategy);
- Influencing factors, which affect the innovation process and cannot be controlled by the corporation (organisational capabilities, the outside world, and the enabling sciences).

Figure 1 illustrates this model as a relationship model and not a linear process. This means that ideas and concepts can iterate and move back and forwards across the five key elements, like the circular shape and the arrows between the key elements suggest.

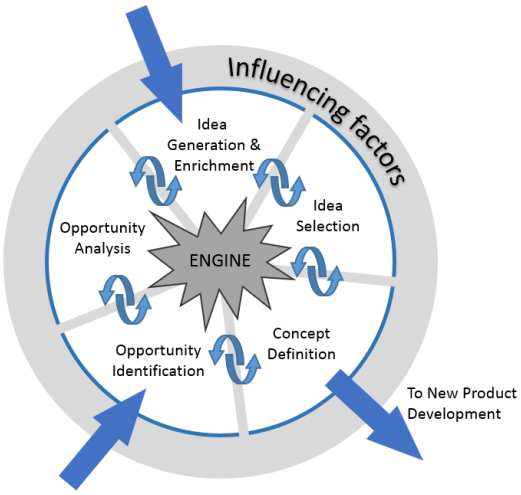


Figure 1 - The new concept development (NCD) (Koen et al., 2001)

Furthermore, the engine "represents senior and executive-level management support and powers the five elements of the NCD model" (Koen et al., 2001). The engine and the five key elements are influenced by the base of the circle, the influencing factors. Finally, the arrows indicate that projects begin at *Opportunity Identification* or *Idea Generation & Enrichment* but only leave the model after the *Concept Definition*.

"The influencing factors are the corporation's organisational capabilities, customer and competitor influences, the outside world's influences, and the depth and strength of enabling sciences and technology" (Koen et al., 2001).

The engine consists of leadership, culture and business strategy and "sets the environment for successful innovation" (Koen et al., 2001).

2.2 Opportunity identification

This element has the objective of identifying opportunities that might be pursued. They can be a "possibility to capture competitive advantage, or a means to simplify operations, speed them up, or reduce their cost" (Koen et al., 2001). Opportunity identification may come from an individual that recognises an unmet customer need or a problem to be solved.

One of the main techniques used to identify opportunities is technology trend analysis, which consists of gathering information regarding technological trends and defining opportunities of process or product improvement that may arise from it. This was the technique used in this work.

An Increasing trend in microservices migration

Like defined in the context section of the previous chapter, there is a clear technology trend in migrating monolithic systems to the microservices architecture. This happens because over time the monolithic system size increases and becomes harder to manage. This mainly affects the maintainability and scalability of the system but also has an impact on the software development lifecycle and on organisational flexibility as the time needed to release new features increases. The microservices architecture suggested by Martin Fowler in 2014 (Fowler and Lewis, 2014) is a commonly chosen proposal to solve these problems. However, the migration of a system to the microservices architecture may cause issues that have costs for the companies that want to follow this architectural refactor. Nonetheless, the benefits that the microservices architecture provides already proved to be worth the migration costs with reference companies in the software engineering field like Amazon and Netflix migrating their systems to this architectural style and evangelising it across the industry.

Therefore, an opportunity is identified in this migration process. If the migration costs or the costs of the microservices architecture itself are reduced, then the microservices architectural style becomes a more appealing solution for companies. It enables them to increase the maintainability and scalability of their systems while reducing the time-to-market of new features implemented in their software systems, at a reduced cost.

2.3 Opportunity analysis

This stage of the NCD model has the objective of analysing the identified opportunity to confirm its viability. For that, additional information is required so that the opportunity identified can be defined as a specific business and technology opportunity. This involves making early and often uncertain technology and market assessments. The technique used may be the same used on the opportunity identification stage, but while it was used with the objective to determine if an opportunity existed, now more resources are expended so that the opportunity is defined with further detail to verify its appropriateness and attractiveness (Koen et al., 2001). The opportunity identified in the previous section is therefore analysed so that it is possible to understand it better and the possibilities of value it may provide.

Since the official definition of microservices, there are multiple reports of migration processes, systematic literature reviews regarding the subject and studies of best practices and patterns for the microservices architecture style and for the migration process. These documents report multiple common problems that still have no explicit or linear solution. They include fundamental intrinsic issues of the microservices architecture style, like dealing with distributed

transactions across microservices and data synchronisation and consistency across multiple databases. Issues of distributed systems, for instance, network-related problems, are another inherent problem of the microservices field. The lack of clear guidelines for the migration of monolithic systems to the microservices architecture is also reported, along with unexpected complexity on the start of the microservices development. Furthermore, it is stated that it is hard to have uniformity across microservices structures and that there is a need to have high levels of automation on the deployment and testing processes of the microservices architecture.

This analysis defines multiple technical challenges that may be addressed regarding the microservices architecture, pre and post-migration from a monolithic system. All of them constitute an opportunity that can bring value to the customer.

2.4 Idea Generation and Enrichment

This key element of the NCD model may be a formal process with the objective of generating new or modified ideas for the identified opportunity. It consists of "the birth, development, and maturation of a concrete idea." (Koen et al., 2001).

"Ideas may be generated by anyone with a passion for a particular idea, problem, need, or situation." (Koen et al., 2001).

On this work, the brainstorming technique was used in order to generate and enrich ideas for the identified opportunity. From the brainstorming sessions, the following enumerated ideas were made. These ideas contribute to the objectives of this work by defining approaches to mitigate possible challenges of microservices migrations or by solving specific issues of this process.

1. **Migrate a monolithic system to a microservices architecture.** The objective of this idea is to identify the problems of this kind of migrations through practical experience, defining solutions for the challenges faced.
2. **Define a technical guide with best practices, conventions, and guidelines for microservices migration.** This idea has the purpose of defining technical guidelines to avoid some of the common problems of the microservices architecture migration process, or at least reduce their impact and costs.
3. **Implement a solution to facilitate the distributed transactions management in a microservices architecture.** On this idea, the focus is narrowed to a specific issue that should be solved and studied with more detail.
4. **Use static analysis to inspect the existent monolithic system and generate suggestions for the boundaries of each of the microservices to be developed** – This idea provides a tool to automatically define the boundaries of each one of the components of the microservices architecture to be implemented, based on the existent monolithic system.
5. **Use model-driven software engineering (MDSE) to create the microservices architecture system based on a defined metamodel** – This idea uses the MDSE

approach to generate a skeleton of the microservices architecture providing a typical structure for all the components of the microservice architecture.

6. **Develop a framework or tool to implement automated integration tests between the different components of the microservices architecture** – The framework or tool developed has the objective of simplifying the process of developing automated tests in the microservices architecture.

2.5 Idea Selection

Idea Selection is the element of NCD where the idea with the most value is selected. This process is affected by insights from the influencing factors and directives from the engine (Koen et al., 2001). After the idea has been selected, further effort will be invested in pursuing and defining it with more detail. The selection process can be just an individual choice between many self-generated options. Usually, this stage is sustained by early personal judgements, with only the idea itself to consider and without more information. Some techniques traditionally used on this process and applied on this work are technical success probability and the strategic fit.

2.5.1 Analytic Hierarchy Process (AHP)

Analytic Hierarchy Process (AHP) is a method to help on the decision-making process, developed in 1980 by Thomas L. Saaty. In order to explain complex decision-making problems, the method models the problem into hierarchical elements. The hierarchy levels are the primary objective, the criteria that define a right decision, and the alternatives that are being considered (Ulkhaq et al., 2018). Therefore, this method was chosen to select the idea that brings the most value from the alternatives described in the Idea Generation and Enrichment section.

Following the AHP method, the hierarchy tree presented in Figure 2 was developed. The first layer defines the main objective of this work that the selected idea should help achieve. Furthermore, the middle layer consists of the following criteria used to evaluate each one of the ideas and choose the best one accordingly.

- Time Restrictions – If the idea presents time restrictions as this work has a pre-defined due date and is limited by it.
- Infrastructure Restrictions – Mandatory Infrastructure requirements for the idea success. The infrastructure available for this work is limited.
- Current Relevancy – Current value for the stakeholder. One of the objectives of this work is to increase the knowledge of the current state of the microservices architecture research, practice, challenges and needs
- Technical success probability – If the idea is achievable with the restrictions of this work with a high success probability.

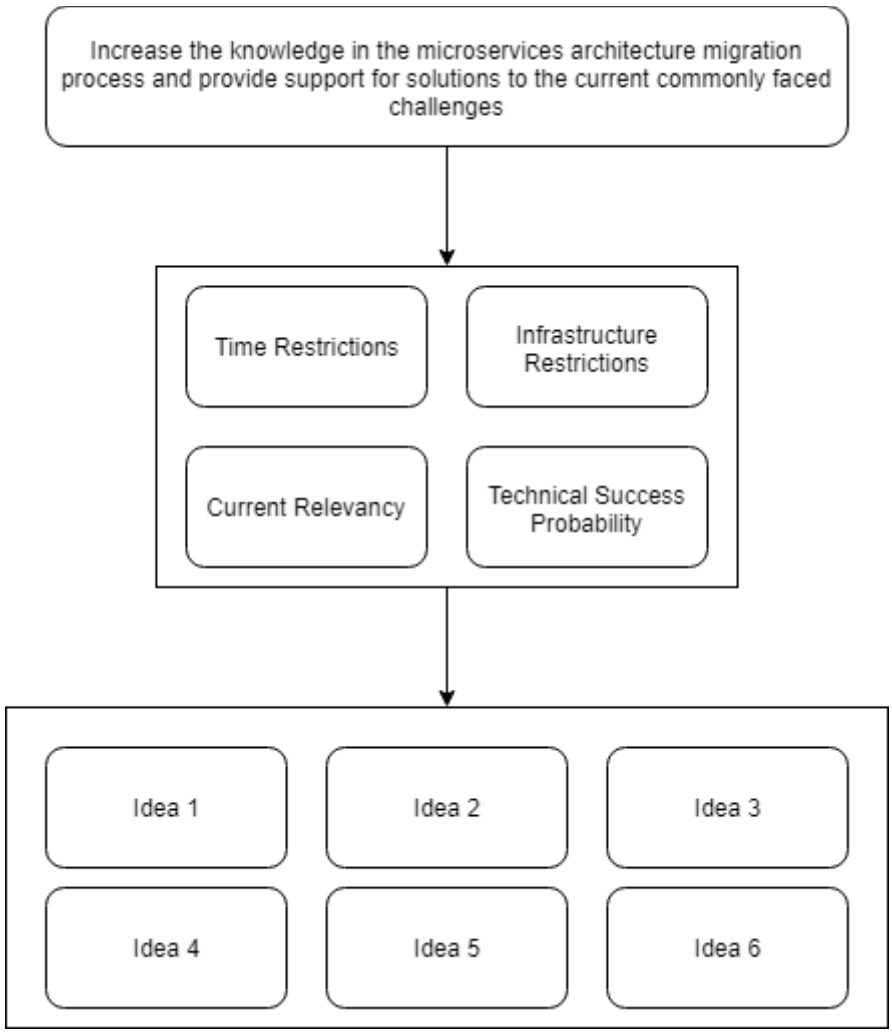


Figure 2 - AHP hierarchical model tree

Finally, on the lowest hierarchical level, the six ideas described in the Idea Generation and Enrichment section are presented.

Based on these criteria, it is possible to evaluate and select the best idea to achieve the main objective. Table 1 below describes the considered weight for each of the criterions following the AHP scale. To accomplish the primary goal of this work, it is essential that the idea is currently relevant and presents the current value. Given the type of project described in this document, there are also some time and infrastructure restrictions that should be considered, as well as the probability of technical success of the chosen idea.

Table 1 - AHP evaluation table

Evaluation Criteria	Time Restrictions	Infrastructure Restrictions	Current Relevancy	Technical success probability
Time Restrictions	1	2	0.33	0.50
Infrastructure Restrictions	0.50	1	0.25	0.33
Current Relevancy	3	4	1	3
Technical success probability	2	3	0.33	1
Sum	6.5	10	1.91	4.83

After defining the weight of each criteria using a pairwise comparison on the table above, the matrix must be normalised to retrieve the priorities of each measure by calculating the mean value of each row. To generate the normalised matrix each cell should be divided the total of the correspondent column. This is presented in Table 2.

Table 2 - AHP normalized matrix

Evaluation Criteria	Time Restrictions	Infrastructure Restrictions	Current Relevancy	Technical success probability	Mean
Time Restrictions	0.154	0.2	0.173	0.104	0.158
Infrastructure Restrictions	0.077	0.1	0.131	0.068	0.094
Current Relevancy	0.462	0.4	0.524	0.621	0.501
Technical success probability	0.308	0.3	0.173	0.207	0.247
Sum	1	1	1	1	1

Calculated the normalised matrix, it is possible to define the priorities of each criterion for the process of idea selection. This is defined in Table 3 below.

Table 3 - AHP criteria priorities

Priority	Criterion	Rate
1	Current Relevancy	50.1%
2	Technical Success Probability	24.7%
3	Time Restrictions	15.8%
4	Infrastructure Restrictions	9.4%

The AHP method concludes that the current relevancy is the most important criterion to apply while selecting the idea with the most value. Technical success probability comes after, followed by time restrictions and then infrastructure restrictions. Therefore, we can now analyse the described ideas based on these priorities to select one.

1. **Migrate a monolithic system to a microservices architecture** – There are multiple reports of migrated monolithic systems, and most of the migrations did not solve any problem or largely increased the knowledge of the community. Therefore, this idea is not considered currently relevant. Furthermore, it would require high infrastructural resources which are not available for this work.
2. **Define a technical guide with best practices, conventions, and guidelines for microservices migration** - This idea may be considered currently relevant as there is an evident lack of guidelines on microservices migrations reported on the literature. However, it may not respect the time restrictions of this work. Furthermore, this is a highly sophisticated solution that may not be able to achieve high technical success.
3. **Implement a solution to facilitate the distributed transactions management in a microservices architecture -** This idea is currently relevant as it solves a recently reported issue and can be designed to achieve a high technical probability of success and respect the time restrictions. It may present some difficulties regarding infrastructure restrictions, but they can be surpassed.
4. **Use static analysis to inspect the existent monolithic system and generate suggestions for the boundaries of each of the microservices to be developed** – There are multiple solutions and studies regarding this topic. Therefore, this idea is not considered currently relevant as it has already similar solutions on the market.
5. **Use model-driven software engineering (MDSE) to generate the microservices architecture system based on a defined metamodel** – This idea is currently relevant as it solves some of the reported problems using a different approach. However, the technical success probability of this idea may be hard to measure as it is a disruptive idea with some uncertainty level.
6. **Develop a framework or tool to implement automated integration tests between the different components of the microservices architecture** – Similarly to idea 4. There are multiple solutions to perform automated integration testing on the microservices architecture. Therefore, this was not considered a currently relevant idea.

Therefore, following the analysis and comparison of each one of the ideas, the selected plan to achieve the objectives of this work is idea 3 - **Implement a solution to facilitate the distributed transactions management in a microservices architecture.**

2.6 Concept Definition

This project has the purpose of identifying the current state of microservices architecture adoption. This study should provide a list of the most common problems since the year this work started (2018). The research can use methods like literature review, and industry surveys or interviews. Therefore, the main requirements are an increased knowledge of the microservices architecture and a catalogue of common challenges and best practices in microservices adoption. Also, the implementation of a solution to manage distributed transactions should be provided. It should be reusable by multiple teams, providing a generic and abstract approach that can be adapted to any microservices oriented system with reduced costs.

The value of the concept defined above will be described in more detail in the following sections where a business model canvas of the solution is presented.

2.6.1 Value proposition

As mentioned in the previous sections and chapters, based on various public documents, there is a trend of companies migrating their systems to a microservices oriented architecture to be more flexible. This flexibility is related to their capacity of adapting to environmental changes or business needs (organisational agility) with inferior costs, which can be achieved with a microservices oriented architecture as it improves the maintainability of the system, as explained before. Furthermore, one of the advantages of microservices is having more flexible scalability and optimised infrastructural costs.

However, even with companies which are references in the software development industry, like Amazon and Netflix migrating their systems to the microservices architecture, there are still some significant issues in this process as related on public documentation and the answers of an industry questionnaire developed and analysed in this project.

This work intends to help solve this problem by analysing and compiling all the issues reported, identifying the most common ones and finding the best solutions for them. Furthermore, this work defines a clear separation between avoidable problems and intrinsic problems of this process that cannot be avoided, but which impact can be minimized.

Also, distributed transactions, which is one of the main challenges, will be further detailed and addressed to implement a solution and reduce this issue when migrating to the microservices architecture.

Therefore, with this work, companies will be able to migrate to microservices architectures with reduced costs. Also, the final system may be better engineered than without the knowledge generated by this work, and for that reason, this work can improve the maintainability, performance, reusability, and other characteristics of the system, which leads to a more resilient system which can be easily adapted to future business needs. Additionally, a clear

solution for distributed transactions will be provided so teams that want to implement microservices architecture will have one less challenge to face.

In order to present this idea in a more structured way, the following Canvas model was developed. Therefore, analysing the model shown in Table 4, stakeholders can find the answers to some business questions like for example Key Partners, Key Activities, Key Resources, Value Proposition, Customer Relationships, Channels, Customer Segments, Cost Structure, and Revenue Streams.

Table 4 cost structure and revenue streams sections clearly show the main reason for the value of this solution. There are almost no costs on using the developed solution, but there are many benefits like providing more resilient systems with higher maintainability, reusability, and increased performance while reducing the overall infrastructural costs of the system. All these benefits are related to the correct use of the microservices architecture and avoiding the main problems of migrating monolithic systems to microservices.

Table 4 – Business model canvas

<table>
<tr>
<td rowspan="2">Key Partners

-Google Scholar, ACM, IEEE, and other digital libraries.

-The industry professionals that answer the questionnaires.

-Companies willing to participate in the study, both through answers to questionnaires and providing support to experiments.</td>
<td>Key Activities

-Systematic Literature Review to identify common problems and solutions.

-Industry questionnaire to confirm the literature review findings.

-Design and implementation of a solution for the distributed transactions issue</td>
<td rowspan="2">Value Propositions

-Identification of the currently most common challenges faced adopting a microservices architecture. This allows companies to avoid or at least be aware of these problems, leading to an overall better microservice oriented final system.

-The solution provided to the distributed transactions challenge will also reduce the costs of microservice architecture adoption as there will be fewer problems to address.</td>
<td>Customer Relationships

-Industry questionnaires identifying the most common problems and evaluating the final solution viability and quality.

-Implementation of the final solution in an interested company.</td>
<td rowspan="2">Customer Segments

- Companies that are interested in performing a microservice migration or solving problems that they are currently facing on microservice oriented systems.</td>
</tr>
<tr>
<td>Key Resources

- Public documentation regarding microservices architecture and migrations.
- Industry knowledge of the field</td>
<td>Channels

Digital Libraries, Technology blogs, Technology conferences, Companies presentation</td>
</tr>
<tr>
<td colspan="3">Cost Structure

- The knowledge provided by the study has no costs.
- The solution developed may require some infrastructural costs to be used.</td>
<td colspan="3">Revenue Streams

- Reduced technical debt;
- Reduced migration time;
- Reduced infrastructure costs;
- Possibly faster development of new features;
- Solution to some of the most common problems;</td>
</tr>
</table>

3 Background

This chapter presents key concepts related to microservice architecture (Section 3.1) and software refactoring (Section 3.2) which are important for the correct understanding of the rest of the document.

3.1 Microservices architecture

As mentioned before, Microservice, as a concept, was first discussed at a workshop near Venice in May 2011, where the participants used the term to describe a typical architectural style that they had been exploring. In May 2012, the group decided to keep the name and started to spread the new architectural style they had defined on different conferences and case studies. Some of the creators of the concept are Martin Fowler, James Lewis, Fred George, Adrian Cockcroft, among others (Fowler and Lewis, 2014). The architectural approach started to get followers at a fast pace, but it was after the publication of Martin Fowler and James Lewis regarding the topic on grey literature that more articles and case studies started to appear and the adoption of the concept increased (Pautasso et al., 2017a, p. 1).

The following definition of the concept, defined by Sam Newman and detailed in his book "Building Microservices" in 2015, will be the one used in this document: "Independently deployable services that work together, modelled around a business domain".

A microservice is an independent component that can be deployed in isolation. However, a microservice alone presents no value which leads to the concept of microservices architecture: "A microservice architecture is a distributed application where all its modules are microservices" (Dragoni et al., 2017). Therefore, a microservices oriented system consists of a distributed application in which its behaviour depends on the communication, composition, and coordination of its microservices via messages (Dragoni et al., 2017).

Microservices are one of the latest trends in software architecture, an evolution of the older concept of Service Oriented Architecture (SOA), but while SOA relies on heavyweight

middleware like Enterprise Service Buses (ESB) or SOAP WSDL (Web Services Description Language), Microservices rely only on simpler technologies, like REST (Representational State Transfer). Furthermore, SOA is usually viewed as an integration solution of already existent systems, while microservices are typically used to develop new and individual software systems (Jamshidi et al., 2018). As a curiosity, before the term “microservices” was established in 2011, similar architectural styles were designed with different names being used. Netflix, for example, used the name “Fine-Grained SOA” for their initial implementation of a similar architecture (Dragoni et al., 2017).

With that in mind, microservices architecture defines that every microservice should have specific and individual responsibility and are generally small and simple systems without significant complexity. Furthermore, they are independently executable systems accessible through a well-defined network interface and only deliver business value when executing together with other microservices, forming a more complex final system (Larrucea et al., 2018).

This strategy increases software agility as each microservice can be independently deployed, versioned, scaled, operated or even replaced without affecting any other service, as long as the network interface is not changed (Jamshidi et al., 2018).

3.1.1 Benefits

Microservices independence emphasises loose coupling and high cohesion concepts, offering different benefits to companies (Dragoni et al., 2017).

Flexibility

The modularity of a microservices oriented system allows an organisation to keep up with changes in the business environment, providing high flexibility on the modifications necessary for it to stay competitive on the market (Dragoni et al., 2017). This type of systems is designed to have high independence and bounded context between its components, leading to high maintainability while being able to add new features (Dragoni et al., 2017).

Maintainability

By definition, a microservice code is restricted to a single responsibility. Therefore, it is easier to understand it than in a monolithic architecture. IDE’s can quickly load the code, and the build is lighter than in a traditional monolith. Working with smaller code bases increases development velocity and allows the development teams to have a real idea of the side effects of any modification to the codebase.

Frequent and fast deliveries

As a microservices oriented system is composed of various small software components, it naturally leads to a high number of software releases in which each version is faster as the size of each element is smaller than traditionally. Typically, this is made using lightweight container technologies and DevOps practices, with the deployment pipeline entirely automated, allowing

the team to deliver working software on arbitrary schedules in a matter of seconds (Jamshidi et al., 2018).

Scalability

“Scalability” can be ambiguous. On one side, it can be the runtime scalability of the system, referring to the system adaptability to handle a higher volume of requests. Alternatively, it can be the development scalability, regarding the possibility of having multiple engineers working on the software at the same time (Jamshidi et al., 2018). Either way, microservices improves both kinds of scalability as the unit of scaling is each microservice.

Regarding the “runtime scalability”, each microservice can be scaled to support its specific needs, independently of the rest of the system. If the microservice oriented system has a performance bottleneck on a specific microservice, it can be scaled in isolation (Baškarada et al., 2018). On the other side, “development scalability” is also improved as each microservice can be developed, deployed and operated independently by different engineers or different teams, which naturally allows the parallel introduction of new features (Jamshidi et al., 2018).

Technological heterogeneity and team autonomy

Each microservice is intended to be able to be autonomously developed, deployed and executed. Therefore, this creates a bounded unit in which the team can make localised decisions – for example, programming language, database technology, libraries and frameworks, among others. Technology heterogeneity leads to a more scalable organisation, where each team can define its strategy for the evolution of its services (Jamshidi et al., 2018).

Fault tolerance

On a monolithic architecture, a service with bugs can cause problems like performance issues, memory leaks, connection failures or even the complete crash of the application. However, in microservices architecture, only the specific service is affected. Microservices isolate system errors and limit their impact across the system. With a well-designed microservice architecture, errors are separated on a single function and do not propagate to the rest of the system, allowing the distributed system to handle the failure on a more graceful way or even recover from it.

3.1.2 Attention points

Section 3.1.1 made clear that there are some advantages for companies to implement microservices. For that reason, renowned companies like Amazon, Deutsche Telekom, LinkedIn, Netflix, SoundCloud, The Guardian, Uber, and Verizon are migrating their services to a microservice oriented architecture (Larrucea et al., 2018). However, microservices also have some less positive aspects that teams should be aware of before adopting the architectural style.

Monolith decomposition

Migrating a monolith to the microservices architecture is considered less risky than redeveloping an entire system as microservice architecture (Larrucea et al., 2018). However, it is common for teams to face challenges associated with monolith decomposition (Baškarada et al., 2018).

There must be a balance between the isolation of the service responsibilities and the right size of the microservice. Each microservice must have a single responsibility, high cohesion, and low coupling, but it should not be so small that the system becomes too complex (for instance doing a single microservice for each possible single operation). Finding the right granularity for each microservice is one of the biggest challenges of this kind of architecture (Fritzsch et al., 2018).

Continuous monitoring, integration, and delivery

First of all, microservices require continuous architecture monitoring and deployment, versioning and deprecating of the services (Larrucea et al., 2018). DevOps practices can help the teams with this, which is the main reason for the usual association of microservices with DevOps practices. Some practitioners even reported that being able to apply DevOps practices was the main reason for microservices adoption and vice-versa, clearly showing a positive association of both guidelines (Baškarada et al., 2018).

However, if this kind of automations are not well established, the deployment process becomes a bottleneck on the software development process (Baškarada et al., 2018), as the microservices architecture typically has many services to be deployed.

Testing process complexity

The testing processes are more complex than in a traditional monolith (Larrucea et al., 2018). A single microservice can be tested independently in a similar way that a single monolithic system. A microservice is probably even quicker to be tested as it is usually smaller. However, a microservice architecture presupposes various microservices are coordinating between each other, and that is why whenever a microservice is changed, it needs to be tested along with all the other microservices in which it depends or that depend on it. Thus, the testing process is more complicated when comparing to a single monolith. These tests can include unit, integration, system and acceptance testing. However, testing distributed systems is inherently more challenging than testing centralised monoliths because they are less stable and have different possible ways of failure and recovery. Testing all the possibilities is a challenge and practically impossible. DevOps practices suggest automating all these tests to minimise the issue, removing it as a bottleneck on the development process (Baškarada et al., 2018).

State management and data consistency

State management becomes more difficult (Larrucea et al., 2018) when using the microservices architecture.

Following the microservices architecture guidelines, each microservice should have its data store, which means that there is no single source of truth. Furthermore, complex business processes usually require the use of a message broker to establish communication between the different services. As a result, this asynchronous messaging may lead to inconsistencies in individual data stores. Some prototypes have used a shared database between microservices to deal with this problem, but this can be seen as out of pattern with the microservices architecture guideline as each microservice is not entirely independent. For these reasons, there is still some debate regarding orchestration and choreography of the communication between the services. Microservices guidelines define that orchestration should be avoided (opposite to the SOA approach). However, orchestration can be valuable to solve consistency problems when implementing complex business processes (Baškarada et al., 2018).

No silver bullet

As microservices have a big hype around them, they are usually seen as a silver bullet to solve all the problems of monolithic applications. However, their advantages come with many attention points, as explained in this section, which can cause high costs for the teams. Therefore, the teams should evaluate if microservices are the best approach for their system. Alternatives without microservices should not be discarded, and all potential solutions should be compared (Carrasco et al., 2018).

On a set of interviews with 19 software architects, all of them reported that microservices solve most of the problems of monolithic applications. However, they are no silver bullet for all kinds of systems (Baškarada et al., 2018). Figure 3 illustrates the architects' point of view.

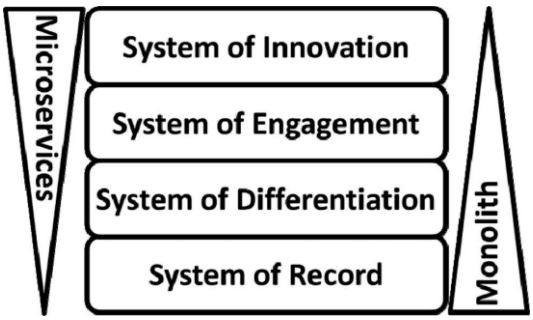


Figure 3 – Types of corporate systems (Baškarada et al., 2018)

One of the main benefits of the microservices architecture is the adaptability to changes in business requirements and the flexibility regarding infrastructural costs. Therefore, they represent a viable solution for systems that have constant changes in business requirements or that require infrastructure support, among other use cases. However, they are not useful for all kinds of software projects, mainly because of the complexity and overhead that they present at the beginning of the development (Baškarada et al., 2018).

Also, some authors recommend starting the development of the software as a monolith and only migrating it later to microservicesto first have a vision of the functional and non-functional

requirements. This knowledge allows the design of a reliable microservices oriented solution – this technique is called Monolith First (Fowler, 2015a).

Furthermore, as stated previously in this section, microservice architecture can typically present some consistency problems or delays. Therefore, if complete consistency is a requirement of the software project, microservices may not be the way to go.

Requires experienced staff

As this approach is recent and presents a relatively high degree of technical complexity, it needs experienced engineers or teams that can learn the technology (Larrucea et al., 2018). The technical complexity derives in significant part from the fact that the microservices architecture is inherently distributed and implementing and sustaining such a system is more complex than monolithic software (Baškarada et al., 2018). The engineers must be aware of the requirements of distributed computing and be ready to implement DevOps practices like continuous integration, delivery, and monitoring. One of the challenges of distributed systems is how to manage distributed transactions, which is an issue addressed by this work.

Distributed systems inherited problems

It also comes with a bundle of issues that are inherited from distributed systems and SOA, its predecessor (Dragoni et al., 2017). Microservices developers and architects need to pay attention to distributed systems challenges and issues, such as:

- Latency is not 0.
- Bandwidth is not infinite.
- The network may not be reliable.
- The network may not be secure.
- The network topology may change.
- The network may not be homogeneous.
- Transport cost is not zero.

For these reasons, all these topics must be considered when implementing a microservices architecture. Therefore, experienced staff is required, and automated development processes can be helpful to assure that microservices development does not become harder than monolithic system development (Baškarada et al., 2018).

3.1.3 Consistency, availability and partition-tolerance

As previously described, microservices architecture assumes a distributed approach to web services implementation, which usually have consistency, availability and partition tolerance as desired characteristics of the system (Gilbert and Lynch, 2002). However, one of the standard conjectures of software engineering, CAP Theorem or Brewer's conjecture, states that a web service cannot ensure more than two of those characteristics (Fox and Brewer, 1999). It was proposed by Fox and Brewer in 1999 (Gilbert and Lynch, 2002) and proved by Gilbert and Lynch in 2002. The reasoning behind this is that when a network partition happens, consistency can

only be ensured with some degree of unavailability while the partitions are synchronized. On the other side, high availability is only possible to achieve without consistency, as there will be some time where the partitions are not synchronised/consistent. Therefore, it is possible to obtain high availability and consistency at the same time but only if the system is not partitioned (Fox and Brewer, 1999).

Microservices architecture constitutes a partitioned system designed for high availability. For this reason, CAP theorem is usually dealt with using the concept of eventual consistency (Stricker et al., 2018). Consistency refers to how and when a client can see updates made to a specific record of a storage system (Vogels, 2009). The author mentions two main types of consistency:

- **Strong consistency**: After the update completes, any observer that accesses the object will obtain the updated value.
- **Weak consistency:** It is not ensured that when an object is updated, all the following accesses will retrieve the updated value. Before that, some conditions need to be met. Inconsistency window is the name used to describe the period between the update and the moment when any access will return the updated value.

Eventual consistency is a type of weak consistency. The storage system assurances that "if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme" (Vogels, 2009).

A distributed transaction consists of the communication between multiple services to accomplish a single business transaction. Using eventual consistency, due to the communication processes referred to as distributed transactions, the various microservices may not be consistent at every moment.

3.2 Software refactoring

Refactoring is the process of improving the internal structure of software without altering its external behaviour. It is mainly focused on improving non-functional requirements without affecting the functional ones. It can be applied on different granularity levels, from code refactoring to architectural refactoring. The term "refactoring" was introduced in 1990 by William Opdyke and Ralph Johnson, and nine years later, Martin Fowler published a book containing a catalogue of structural changes that were observed in multiple languages and application domains (Murphy-Hill and Black, 2008). These structural changes are mainly code refactoring techniques like methods or variables renaming, class splitting, among others. Over the years, these techniques got much attention from researchers and practitioners and are currently a mainstream practice of software engineering. There are manual and automatic tools available to support this refactoring (Murphy-Hill and Black, 2008), with some of them being

included in the most recent Integrated Development Environments (IDE). Therefore, code refactoring has been a success since it was first introduced.

However, architectural refactoring (AR) does not have the same support yet. AR consists of activities that have the objective of solving indicators that the current architecture of the system is not aligned with the current requirements and restrictions, which can cause issues regarding maintainability, scalability, and others. These indicators are called *architectural smells*, which is something that naturally happens over time with the incremental evolution of the software (Zimmermann, 2015), and changes in both functional and non-functional requirements.

Therefore, an AR improves one quality attribute of the system without affecting the overall functionality. However, to achieve this, it might negatively influence other attributes, which is a trade-off. The reasoning for this is that an AR revisits individual architectural decisions that were made previously in the lifecycle of the project, but that may not be adequate for the current requirements and restrictions. For these reasons, an AR selects alternative solutions for architectural issues in order to achieve the current requirements and restrictions, which may be different from the ones analysed when defining the initial architecture of the system. After analysing the current requirements and deciding alternative solutions to the identified design issues, the required changes must be implemented and documented. Contrary to code refactoring, the implementation tasks of ARs usually refer to structural changes such as dealing with components, subsystems and their interfaces. Some patterns in this kind of refactoring can be identified, and Architectural Refactoring Catalogues emerge. Figure 4 represents an example of an architectural refactoring catalogue. Migrating a monolithic system to a microservices architecture can be considered an AR.

Viewpoints, types of ARs, and the associated ARs.

Viewpoints	Types of change		
	Elaboration ARs	Adjustment ARs	Simplification ARs
Functional	Split Component Responsibility	Expose Internal Feature as Component Responsibility	Merge Component Responsibilities
	Shift Responsibility to New Component	Shift Responsibility to Existing Component	Merge Components
	Split Layer (Move Components to New Layer)	Replace Layer	Join Adjacent Layers (Collapse Layers)
Concurrency, Information	Distribute Processing (Introduce Concurrency)	Change Distribution Algorithm (for example, from Round Robin to Priority Driven)	Consolidate Processing (Remove Concurrency)
	Introduce Cache	Change Cache Entry Lookup Key	Remove Cache
	Prepopulate Cache (Load More Eagerly)	Change Cache Cleanup Strategy	Start with Empty Cache (Load Lazier)
Deployment	Assign Logical Component to New Deployment Unit	Change Scaling Strategy (for example, from vertical scale-up to horizontal scale-out)	Merge Deployment Units
	Split Deployment Unit (to deploy on separate existing or new nodes)	Move Deployment Unit (from one server node to another)	Consolidate Nodes
Deployment, Operational	Factor Out Node into New Tier	Split Tier	Collapse Tiers
	Introduce Clustering	Change Load-Balancing and Failover Policy	Remove Clustering

Figure 4 - Architectural refactoring catalogue example (Zimmermann, 2015)

4 State of the art

In order to define the best possible solution for the problem at hands, it is essential to understand what is known or exists concerning the topic. Therefore, it is crucial to understand what can be improved in the field or what has not been addressed yet. This chapter discusses this and is divided into two sections, one for each objective of this work.

4.1 Microservices migration research

This section has the main aim of studying the literature to identify existent approaches for microservice migrations research.

As previously mentioned, microservices architecture was officially introduced to the community in 2014. Since then, research on the microservices migrations field has been evolving. In 2016, a systematic mapping study identified three articles addressing the microservices migration subject. One year after, in 2017, Di Francesco et al. pointed 16 studies approaching the topic. Therefore, research on the matter is still evolving, and Microservices migration is referred to as a future trend (Fritzsch et al., 2018). On this topic, some studies on microservices migrations will be analysed and compared to identify what can be improved in this research field.

4.1.1 Existent approaches

The following approaches were identified in a research performed in February 2019. Therefore, only documents or articles published between 2014 and February 2019 were considered.

"Microservices migration patterns" - (Balalaie et al., 2018)

The *"microservices migration patterns"* is an empirical study developed in 2017 and published in 2018, focused on identifying the most common patterns on microservices migrations. The study proposes a list of microservices migration patterns using a metamodel template defined in the same work. The identified trends are obtained from the personal experience of the

authors and have been identified from empirical research of industrial scale microservices migration projects. All the patterns refer to the migration planning phase of the migration. A method to combine different patterns into a migration plan is also proposed.

“Microservices: A Systematic Mapping Study” - (Pahl and Jamshidi, 2016)

The work of Pahl and Jamshidi addresses the application of microservices as an option to migrate services to a cloud computing infrastructure. This systematic mapping study published in 2016 analyses 21 selected studies which were published until the end of 2015 and since the emergence of the microservices architecture. The study aims to analyse existent literature to answer the following questions.

1. What are the main practical motivations behind using microservices?
2. What are the different types of microservice architectures involved?
3. What are the existing methods, techniques and tools to enable microservice architecture development and operation?
4. What are the current research issues, and what should be the future research agenda?

“Architectural Patterns for Microservices: a Systematic Mapping Study” (Taibi et al., 2018)

The objective of this work performed in 2018 is to analyse reports of microservices usage to extract from those use cases the used microservices architecture patterns and principles. The systematic mapping study presents identified common microservices patterns on a catalogue using a defined template format that summarises the advantages, disadvantages, and lessons learned for each of the designs. The study also describes some universal guiding principles of the microservices architectural style.

“Migrating towards Microservice Architectures: an Industrial Survey” (Di Francesco et al., 2018)

This 2018 research work consists of an empirical study of microservices migrations practices in the industry. The authors performed several interviews and questionnaires to industry microservices migrations practitioners. The work presents information regarding the completed activities during the migration and the challenges faced. The objective was to identify the recommended future direction on microservices migrations research and the most relevant problems.

“Migrating towards Microservices: Architecture Smells” (Carrasco et al., 2018)

This study analyses both academic literature and grey literature, identifying a total of 58 documents. The main objectives of this 2018 work were to identify architectural bad smells (bad smells is a commonly used term on software engineering when referring to bad practices) present in the microservices architecture and the widely used solutions to avoid them.

4.1.2 Comparison of existent approaches

This section compares the previously described works, analysing the following characteristics of each one:

- Coverage – As microservices is a broad theme with multiple issues to be addressed, these characteristic analyses the topics covered in the described works.
- Publication year – This characteristic points the year in which the document was published.
- Type – The type of study the work consists, which can be a survey, literature review, reports, solutions proposals, and others.
- Objective – This characteristic identifies the primary goal the authors of the work pretended to achieve with their research.
- Approach – The approach used in the study is indicated here.
- Target – The target source of information for the research.

Table 5 presents the described works side by side and the defined characteristics of each one.

The analysed works are a sample of the different kinds of existent research on the microservices field. It is possible to conclude that most of the current research targets the existent literature. One of the studies targets both the literature and the industry. However, it is based on reports and the authors' experience and no direct contact with industry practitioners was considered. The only found work that directly addresses industry professionals does not compare it with information present on literature. It is also the single study found addressing migration challenges, while most of the studies focus on architectural patterns or best practices. *"Migrating towards Microservices: Architecture Smells"* does discuss bad practices of microservices adoption but only to contextualise proposed solutions and guidelines for the correct usage of the architectural style. It is also noted that systematic mapping studies are a common practice for literature reviews regarding this field.

Table 5 – Comparison of previous microservices research works

Name	*"Microservices Migration Patterns"*	*"Microservices: A Systematic Mapping Study"*	*"Architectural Patterns for Microservices: a Systematic Mapping Study"*	*"Migrating towards Microservice Architectures: an Industrial Survey"*	*"Migrating towards Microservices: Migration and Architecture Smells"*
Coverage	Migration Patterns	Microservices State of the art	Architectural Patterns	Migration practices and challenges	Architecture best practices
Publication year	2018	2016	2018	2018	2018
Type	Empirical study of multiple industry projects	Literature review	Literature review	Empirical	Literature review
Objective	Patterns identification	Identifying the existent research on microservices	Pattern identification	Gather industry data for future research	Identify Microservices Bad practices and how to avoid them
Approach	Empirical Research and standards definition	Systematic Mapping study	Systematic Mapping Study	Industry Survey	Traditional Literature review
Target	Literature and Industry	Literature	Literature	Industry	Literature

Therefore, analysing the existent research in the microservices field, the following points were identified:

- Lack of research in microservices challenges. Most studies focus on patterns and best practices.
- Comparison of the information retrieved from literature with industry testimonies is not a main focus of the analysed studies.
- Systematic mapping studies are a common good practice for software engineering literature reviews.

The aim of most of these studies is to identify common patterns or techniques used when adopting microservices, but none focus specifically on identifying the most common technical challenges. Furthermore, they do not cross the information from the literature with industry reports. Therefore, they cannot be used to replace the immediate objectives of this work. The present work will, therefore, address these issues and fill the identified research gaps in the microservices field.

4.2 Distributed transactions

Distributed transactions are one of the main challenges of microservices adoption. When using a monolithic architecture, transactions usually occur in a single database. However, when splitting a monolith into a microservices architecture, these transactions are usually also divided and start occurring in multiple services, creating a distributed transaction. The main issue is to how to manage it across the entire architecture, dealing with issues such as atomicity, isolation of concurrent requests, and consistency (Ntentos et al., 2019). This section describes some existent solutions for this challenge.

4.2.1 Two-phase commit (2PC)

Two-phase commit is a classic technique proposed by Gray in 1978 to handle transactions using two different phases to complete them (Gray and Lamport, 2006). In a distributed system, a transaction is performed between two processes. In the context of the microservices architecture, these processes are the microservices.

In the first phase of 2PC, all microservices lock resources to prepare for a data change. When all the microservices involved are ready, the second phase begins and all the microservices apply the actual changes. The global transaction manager (TM) has the responsibility of coordinating the transaction process and calls the microservices to execute both of the mentioned phases. If a single microservice fails in the first phase, the TM will abort the transaction and rollback all the locked resources (Gray and Lamport, 2006).

One of the advantages of 2PC is that it ensures strong consistency of the transaction. The transaction only ends if all the microservices are updated or if all of them are not modified. Furthermore, the changes are not visible until the TM commits all the changes in all microservices (Xiang, 2018).

To achieve this, 2PC is a blocking protocol as the resources are locked until the second phase ends. This protocol was developed for database systems (Gray and Lamport, 2006), where transactions take around 50ms to complete (Xiang, 2018) and therefore, these locks may not be a problem. However, in the context of microservices, this is usually not the case as the communication is made between different software components (web services). Therefore, 2PC may not be an excellent alternative to perform distributed transactions as it may become a system bottleneck. Furthermore, if two transactions occur at the same time in the microservices architecture, they may lock each other and cause a deadlock (Xiang, 2018).

4.2.2 Saga

Transactions between microservices are propagated across multiple systems, and therefore they are usually slower than a local operation of a single database. This kind of transactions is called long-lived transactions (LLT). Using the 2PC, while the LLT is being processed, other

smaller transactions are delayed. The saga concept was proposed by Garcia-Molina in 1987 and dealt with this issue. An LLT is considered a saga “if it can be written as a sequence of transactions that can be interleaved with other transactions” (Garcia-Molina, 1987). Following this approach, either all the transactions of the saga are successful, or compensation transactions are executed to amend the modifications applied by previous transactions. The compensation is applied in reverse order so that the system returns to its former state, which is useful to understand what is happening in the system in complex multi-transactional processes and also to have a rollback mechanism without blocking the resources.

A compensation action does the opposite of the first operation. However, each change made by a saga activity is not isolated, which means that other assets can use the resources modified by one of the saga transactions before the saga is completed. Naturally, this can cause the compensation transaction to fail and allow other assets to use incoherent data (Greenfield et al., 2003), which is a disadvantage of this approach. Furthermore, as sagas are not consistent at every moment, this technique uses the eventual consistency model described in Section 3.1.3 and are commonly applied in the microservices architecture, as it is used to achieve high availability and partition tolerance. The implementation and debugging of a saga are highly complex, with different challenges to be faced depending on the saga approach used. The alternatives are described in sections 4.2.2.1 and 4.2.2.2.

4.2.2.1 Choreography-based saga

In a choreography-based saga, control is distributed over the different components of the system, and there is not a central orchestrator service, which means that each service listens to the other services events and decides what to do next (Rosa, 2018a).

Following this alternative, a microservice starts the transaction and publishes an event. The following service in the saga sequence listens to the message and publishes another. The transaction ends when a service listens to the message from the previous transaction participant and does not publish a new event, or there are no subscribers of the last event. Figure 5 illustrates this mechanism.

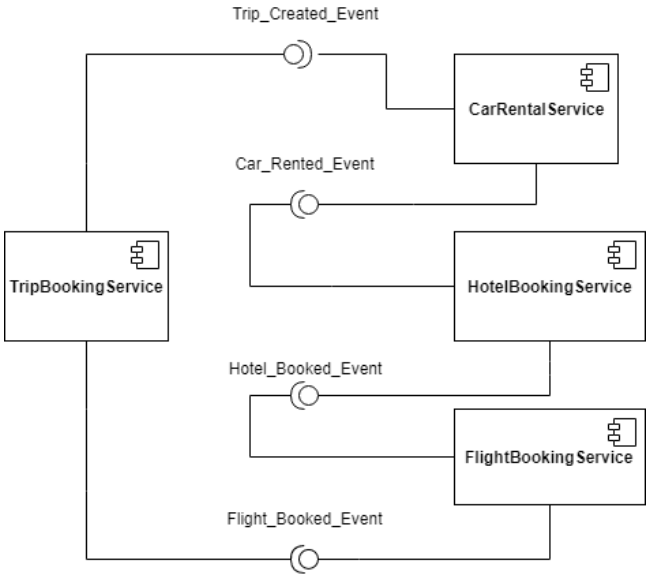


Figure 5 – Successful choreography example

The case of a trip booking system was used. The trip is initially created in the *TripBookingService* in a *pending* state. The trip-booking will only be complete when a car, a hotel and a flight are booked. Therefore, the *CarRentalService* listens to the *Trip_Created_Event* and rents a car, publishing the *Car_Rented_Event.* Similar processes happen for hotel and flight bookings. When all are completed, *TripBookingService* listens to *Flight_Booked_Event* and updates the trip state to *complete*.

In this example, the bookings are made in sequential order. An alternative would be to have all the services (*CarRentalService*, *HotelBookingService* and *FlightBookingService)* listening to *Trip_Created_Event* in parallel. In this scenario, the trip state would be set to *complete* by *TripBookingService* when it had consumed all the required events (*Car_Rented_Event Flight_Booked_Event,* and *Hotel_Booked_Event).*

The described executions are successful scenarios. However, in Figure 6, a situation where the transaction failed is illustrated.

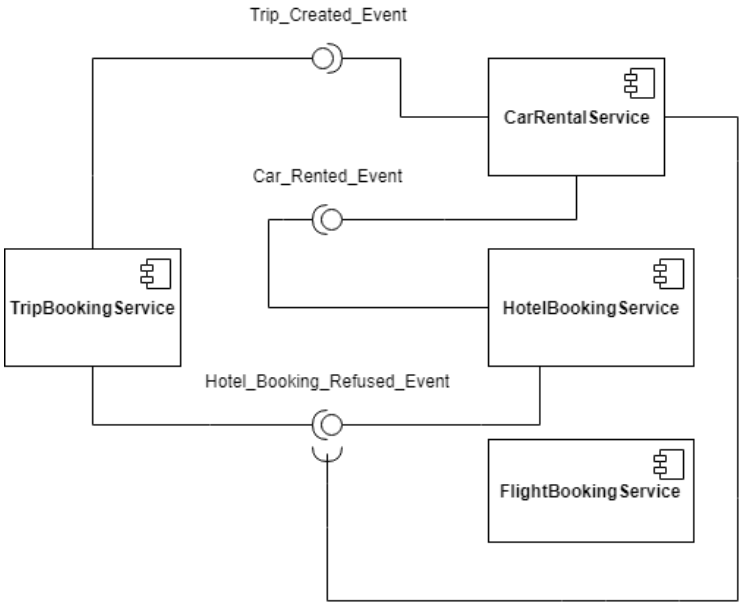


Figure 6 – Failed choreography example

The saga pattern suggests that when a failure happens, the transaction must be roll-backed using compensation actions in reverse order. A compensation action does the exact opposite of its correspondent action. These compensations must be applied in reverse order to the standard transaction sequence to revert the events by the same procedure they occurred and reset the system to its original state. In the example above, the *HotelBookingService* failed to book a room, and therefore, a *Hotel_Booking_Refused_Event* was published. The other services are ready to listen to this event to apply compensation actions to return to their original state. *CarRentalService* must cancel the rented car, and the *TripBookingService* must set the trip state as *failed. FlightBookingService* does not perform any action in this scenario, as *Hotel_Booked_Event* was not published.

It is essential to define a transaction identifier (correlation ID) so that the different events can be correlated to the same transaction, which is crucial to apply the compensation actions to the right resources and to use monitoring practices to visualise process execution across the multiple services.

The described approach to choreography-based saga is event-driven. However, there is also the possibility of using the routing slip pattern to achieve this. With this approach, no events are published. Instead, a routing slip containing all the steps of the transaction is attached to the message. Therefore, each of the services marks its contribution as successful and sends a new message using the same routing slip to the following address specified in the routing slip. If the service is not successful, it marks its contribution as failed and sends a new message using the same routing slip to the previous participant of the transaction to apply compensation actions. When the previous participant receives the message, it applies the compensation action and forwards the issue to the participant before it until the participant that started the transaction is compensated and the system gets back to the original state.

Choreography approach allows all participants of the transaction to be loosely coupled as they do not directly interact with each other; they react to messages. However, when the transaction is complex and has many participants, it can become confusing, and it is difficult to identify what events each service listens – this issue can be mitigated with good monitoring practices. Furthermore, a cyclic dependency can also be created if two services are listening to messages from each other, but this should be avoided.

4.2.2.2 Orchestration-based saga

In an orchestration-based saga, control is centralised in a single service. It is responsible for the management of the saga transaction, including decision making and the business transaction sequence (Rosa, 2018a). Figure 7 presents an example of this approach.

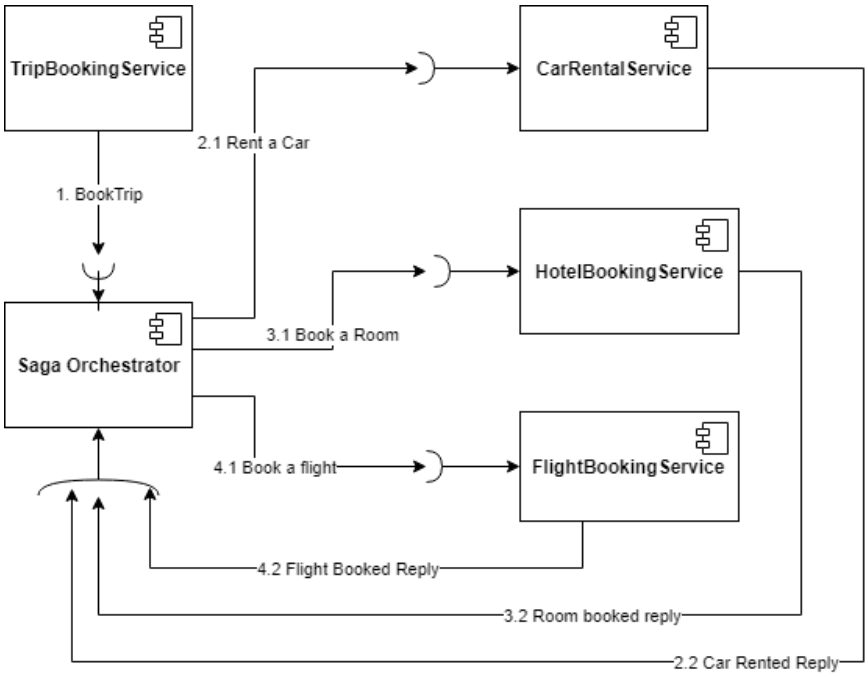


Figure 7 - Successful orchestration example

In this approach, there is a new service in the system. It has the responsibility of managing the transaction and invoking other participants' actions at the right moment following the transaction sequence. This service is called *orchestrator* and knows the transaction flow required to execute the specified request. In the provided example, the *orchestrator* receives a request to book a trip. It recognises that to accomplish the solicitation it has to request *CarRentalService* to rent a car, *HotelBookingService* to book a room and *FlightBookingService* to book a flight. The orchestrator receives the correspondent replies, so it is aware of the success or failure of the different operations to apply compensation actions if needed. Below in Figure 8 is an example of a failure in a transaction using orchestration-based saga pattern.

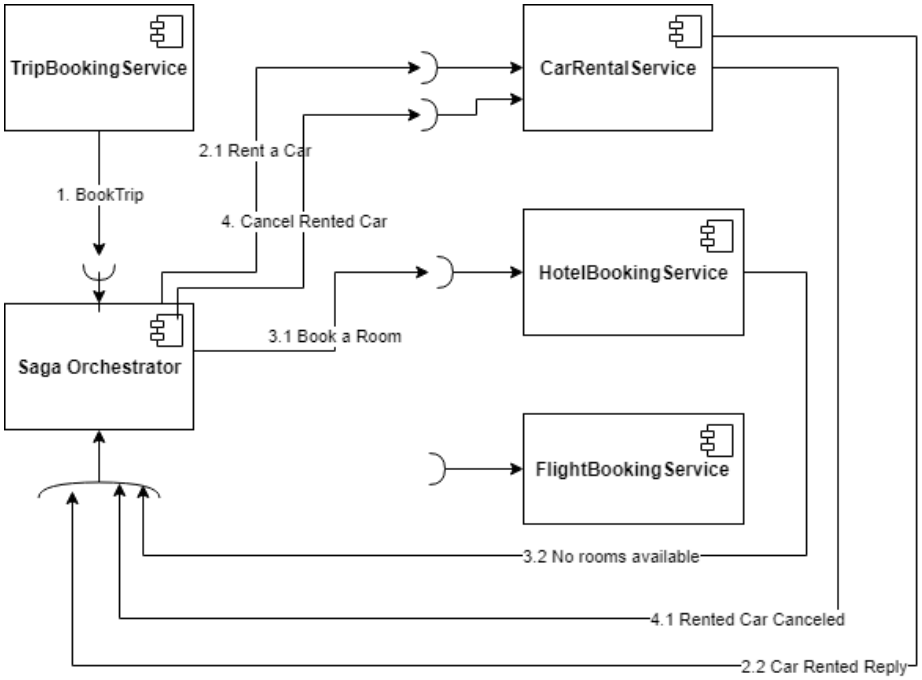


Figure 8 - Failed orchestration example

In this example, the *orchestrator* booked the car successfully, but there were no available rooms to book the hotel. Therefore, when the *orchestrator* received the *No rooms available* reply, it requested the *CarRentalService* to cancel the car it had rented. *CarRentalService* then replied that the rented vehicle was successfully cancelled. If this was not the case, the *orchestrator* could retry the request indefinitely or send a notification to the system administrator for manual intervention.

A commonly used way to implement an orchestrator is applying the State Machine pattern and using workflow automation tools with the business process model and notation (BPMN) to define the transaction flow.

One of the main advantages of this approach is the inexistence of cyclic dependencies between services as the orchestrator invokes the participants, but the opposite does not apply. Furthermore, the distributed transaction orchestration is centralised in a single service and therefore, the complexity does not increase with the number of steps the transaction requires.

One of the risks of using this approach is the concentration of too much logic in the orchestrator, which is something that must be thought of when implementing this pattern. Also, contrary to the choreography approach, using orchestration the infrastructure complexity increases as there is one more service to manage. Finally, the orchestrator service is a single point of failure for all the business processes that it manages.

Comparison

Table 6 compares the key characteristics of each approach. They have different advantages and disadvantages. The choice of the best implementation is entirely dependent on the context in which they are going to be applied.

Table 6- Saga alternatives comparison

	Orchestrated saga	**Choreographed saga**
Coupling	High	Low
Responsibility Segregation	Isolated	Not Isolated
Dependencies Management	Centralised	Decentralised
Infrastructure Complexity	High	Low
Transaction management	Centralised	Decentralised

Regarding coupling, orchestrated saga requires a single component (the orchestrator) highly coupled to all the other services which increase the overall coupling of the system. In the choreographed saga, this is not an issue as the services only subscribe to messages and do not need to be aware of all the other system components.

By following a centralised approach, the orchestrated saga isolates all the responsibility of transaction management in a single component, which also isolates the management of inter-service dependencies, and therefore it is easier to visualise the business process in a single point and modify it. In the choreographed approach, this high-level view can only be obtained by monitoring mechanisms that observe the messages flowing in the system.

Finally, the orchestrated saga introduces one extra service in the system, which increases its complexity and infrastructure requirements. Furthermore, it constitutes a single point of failure for the success of any business transaction. The choreography approach avoids these issues.

4.2.3 Comparison of existent approaches

This section compares the previously described works, analysing the following characteristics of each one:

- Approach – The procedure used to ensure distributed transactions.
- Consistency model – The way the approach deals with consistency.
- Resource Isolation – The way that resources are managed during the transaction execution.
- Rollback mechanism – Fail recovery strategy used.
- Use context – In what context the technique is usually applied.

Table 7 presents the described implementations side by side and the defined characteristics of each one.

Table 7 - Comparison of distributed transactions implementations

	Two-Phase Commit (2PC)	Saga
Approach	Locks resources until all participants are ready to apply modifications. If at least one fails to prepare, the resources are unlocked in its previous state. Otherwise, the modifications are applied.	Saga applies the modifications in every resource sequentially. If one execution fails, the modifications are reverted applying compensating transactions in reverse sequential order.
Consistency model	Strong Consistency	Eventual Consistency
Resource isolation	Blocking	Non-blocking
Rollback mechanism	Changes are reverted before being available.	Compensation transactions are executed.
Use context	Short-Lived Transactions	Long-Lived Transactions

As previously mentioned, transactions between microservices are applied over the network and sometimes over multiple software components. For this reason, they are usually Long-Lived Transactions. Furthermore, microservices are, by definition, a distributed system designed to ensure high availability. Therefore, following the CAP theorem, they are unable to provide strong consistency. Usually, microservices architectures implement the eventual consistency model.

In the microservices context, to implement 2PC, some degree of availability must be lost, and some services may be locked for a relatively long time, which may affect the overall performance and availability of the system.

Following these requirements and analysed the existent solutions, saga pattern presents a better alternative to manage long-lived transactions in an environment where eventual consistency is acceptable, like when using the microservices architectural style.

4.3 Related technologies

At a conceptual level, there are solutions which were analysed in Section 4.2. However, it is also important to realize the available implementations and their advantages and disadvantages.

4.3.1 Two-phase commit (2PC)

There are some specifications and implementations of this technique. Starting with XA Transactions, it is a specification of a protocol to implement 2PC that coordinates single transactions that require access to multiple distributed resources, defined by Open Group in 1991. The specification ensures that any modification is committed in every affected resource. Otherwise, all the modifications are fully rolled back (Open Group, 1991). The protocol defines

interfaces that should be followed to accomplish that mechanism. The transaction manager (TM) or XA Coordinator manages the global transactions, and all the resources affected should be enlisted in the TM.

Furthermore, the TM executes methods exposed by the resources. The methods exposed manage the resource through a Resource Manager (RM). The RM is responsible for managing a particular resource such as a database. This way, the methods “prepare” and “commit” of the 2PC previously explained are exposed by the resource and invoked by the TM in order to ensure the ACID properties of transaction that access multiple resources.

Regarding the application side of the XA Protocol, the Java Transaction API (JTA) defined by Sun Microsystems is a Java implementation of the specification. It consists of a high-level API to facilitate the use of XA Transactions using Java software (Kosaraju, 2007). It consists of three main components: a high-level java interface that defines the transaction boundaries, a Java mapping of the main specification parts such as the XA resource, and finally JTA defines a Java interface to ease the implementation of the transaction manager allowing Java applications to manage transactions with 2PC. This java interfaces defined by JTA are implemented by different frameworks, such as JBossTS, Atomikos and Bitronix JTA.

Finally, regarding the resources involved in a distributed transaction like databases or messaging systems, there are also implementations for XA Transactions protocol. For instance, MySQL database provides an implementation of the 2PC protocol. Also, Apache’s ActiveMQ provides ways to follow the specification.

4.3.2 Saga pattern

This section presents some of the technologies used to implement the saga pattern. The set of solutions analysed is heterogeneous, consisting of enterprise products, open-source libraries, and even cloud provider services.

4.3.2.1 NServicebus

NServicebus is a framework for .NET systems that provides messaging and workflow management features. The licensing is only free for personal use with paid options for commercial use. It has a relevant usage in the microservices ecosystem, and one of its features is the implementation of saga.

The framework isolates all the logic of saga state persistence in a single generic class that can be inherited to define the saga steps and actions. Therefore, applications using the framework only need to define the data to be persisted as the saga state and define what actions should be executed in each step of the saga. The framework persists and manages the saga state. The framework is message-driven and does not support HTTP messages.

A possible drawback is that it does not provide an explicit implementation for compensating actions, except for a timeout feature when sagas are not completed in a given period. Also, in order to use the saga implementation, other features of NServiceBus must be used, which can

require the refactoring of the system code if the framework is used in an existent service (Particular, 2019).

4.3.2.2 Eventuate Tram

Eventuate Tram is a framework for Java systems that provides a way to send and receive messages as part of a database transaction, ensuring that an application can atomically update the database and publish messages.

Eventuate Tram Saga is the saga implementation of Eventuate Tram specifically for microservices that use JDBC or JPA. Eventuate Tram Sagas does provide support for compensating actions. Similarly, as with NServiceBus, in order to use Eventuate Tram Sagas, the entire Eventuate Tram framework must be used, which may not be ideal in some use cases. Also, Eventuate Tram support is paid, but the framework can be used for free (Richardson, 2019).

4.3.2.3 Aws Step functions

Aws Step functions is a tool to help the workflow management process of a distributed system. It is different from the previously mentioned solutions as it is a service and not a framework and allows a serverless implementation of the saga pattern. Aws Step functions provide a visual workflow definition for the coordination of the components of a distributed system. It provides intrinsic retries and error handling mechanisms. Also, it logs the state of each step of the defined workflow, which allows the user to diagnose and debug problems quickly when a specific step fails.

The advantage of this service is that it is technology agnostic and is generic enough to support compensating actions by defining them in the workflow. Aws Step functions are free up to 4000 state transitions but charge a fee for each state transition after that (Amazon, 2019).

4.3.2.4 Camunda

Camunda is a Business Process Model and Notation (BPMN) engine for .NET, which usage is allowed under the Apache 2.0 open source license (Camunda, 2019). It is essential to notice that as saga pattern defines a business workflow or business process, the pattern can be implemented as a state machine – following the orchestrated saga approach. BPMN is a commonly used notation to design this kind of processes. Also, there are multiple BPMN engines to execute the defined BPMN models that can work as a saga orchestrator. This notation provides support for the representation of compensating actions which facilitates the design of a saga workflow to be executed on a BPMN engine. Camunda also provides observability mechanisms to monitor the execution of choreographed saga processes.

4.3.2.5 Workflow Core

Workflow Core is an open-source project which was started in November 2016 by the individual contributor Daniel Gerlag. Over the years, twelve more individual contributors contributed to the project. At least 39 software projects use the library developed by Gerlag. The .NET workflow engine usage is allowed under the MIT open-source license. This project is the only library found for distributed transaction management, opposite to the other described

technologies which are frameworks, workflow servers or cloud services. Therefore, it is the most lightweight of the ones analysed in this section, not requiring the usage of any other feature, framework or technology.

Also, the software provides saga implementations with compensation actions in an orchestrated saga approach. It also provides detailed and helpful documentation for teams to implement the framework in their systems. Workflow Core also supports Conductor, a workflow server also developed by Daniel Gerlag, which provides a workflow server using workflow core as its engine – not related with Netflix Conductor (Gerlag, 2019).

4.3.2.6 Netflix conductor

Conductor is a Java-based workflow orchestration engine developed by Netflix to manage their microservices architecture distributed transactions. It runs in the cloud, providing an API that allows clients to define and orchestrate workflows. The workflows can have compensation actions defined, and the clients can be developed in any technology that can communicate with the provided API. Conductor is free under the Apache 2.0 open-source license (Netflix, 2019).

4.3.2.7 Cadence

Cadence is a workflow orchestration engine quite similar to Netflix Conductor but developed by Uber following the MIT open-source license (Uber Engineering, 2019).

4.3.2.8 Comparison

Table 8 compares the described technologies side by side.

Table 8 - Saga technologies comparison

Name	License	Tech Type	Channels	Saga Approach	Software Type	Source
NServicebus	Free for personal use only.	.NET	Message Driven	Orchestration-oriented	Full Framework	Open Source
Eventuate Tram	Free – Apache 2.0. Provides paid support options	Java	HTTP and Message Driven	Orchestration-oriented	Full Framework	Open Source
Aws Step Functions	Free up to 4000 state transitions. Paid after that.	Cloud	Agnostic	Orchestration	Cloud Service	Closed Source

Name	License	Tech Type	Channels	Saga Approach	Software Type	Source
Camunda	Free – Apache 2.0	Agnostic	Agnostic	Orchestration	Framework	Open Source
Workflow Core	Free – MIT	.NET	Agnostic	Orchestration	Library	Open Source
Netflix Conductor	Free – Apache 2.0	Agnostic	Agnostic	Orchestration	Workflow Server	Open Source
Uber Cadence	Free – MIT	Agnostic	Agnostic	Orchestration	Workflow Server	Open Source

Most of the solutions require structural changes to be adopted, either through the usage of a framework or connecting the system to a workflow server or cloud service. Most of the solutions are free and open source. Also, none of the identified technologies uses choreographed sagas.

The microservices architectural style is an evolution from SOA, and one of the differences is the recommended approach for service integration. In SOA, usually services were integrated using orchestration through a service bus or business process server. However, multiple principles of microservices ("smart endpoints, dumb pipes", "decentralised governance", "decentralised data management") suggest that the centralization of too much logic in a central orchestrator increases the risk of having too much coupling between the services (Fowler and Lewis, 2014). Therefore, one possible reason for choreographed sagas solutions to not be common is that most tools used in microservices are still evolving from the original SOA context.

Another possible reason is that in choreographed sagas there is no central unit managing distributed transactions, and therefore the implementation is directly related to each business process, making it harder to implement in a generic solution.

5 Problem statement

This chapter describes the main problems to be addressed, along with the objectives, contributions and methodology of this work.

5.1 Problem description

Even with renowned companies as early adopters of microservices, there is still few academic research on the subject and empirical research is still limited (Baškarada et al., 2018). Also, the transition to a microservice architecture is *"an error-prone process with deep pitfalls resulting in high costs for mistakes"* (Carrasco et al., 2018). One of the problems identified is using a system decomposition strategy which causes high coupling between the different services and creates a "distributed monolith", having both the limitations of a monolith and a distributed system and none of the advantages of the microservices architectural style (Fritzsch et al., 2018). Another problem with this process is how to deal with multi-tenancy and statefulness (Furda et al., 2018).

In addition, architects and developers face different distributed systems challenges, as well as systems integration difficulties (Ulander, 2017) when transiting to microservices. There is also a more significant infrastructural complexity on the development of a microservice architecture. Development teams need to be knowledgeable about DevOps practices (Baškarada et al., 2018) to mitigate this issue, which is something that companies are often not aware when they decide to migrate to microservices (Carrasco et al., 2018).

Furthermore, depending on the monolithic system, the migration to the microservices architecture can be costly and a long-lasting effort (Fritzsch et al., 2018) for the company. Finally, it is recognised a *"lack of general guidelines for migrating monoliths towards microservices"* (Carrasco et al., 2018) and substantial qualms about technical aspects of microservice adoption have been reported (Baškarada et al., 2018). The literature identifies "a lack of systematic guidance on the refactoring process for existing monolithic applications" (Fritzsch et al., 2018). Some of the publications discussing Microservices also mention the migration process, but it is

not the main focus of the documents. A 2016 systematic mapping study identified 3 out of 21 articles that deal with migration topics. In 2017, Di Francesco et al. pointed 16 out of 71 studies approaching the subject. Therefore, research on the topic is still evolving, and microservices migration is referred to as a future trend (Fritzsch et al., 2018).

In a monolithic system when there is the need to ensure consistency between multiple resources, an ACID (atomicity, consistency, isolation, and durability) transaction is usually used (Hasselbring and Steinacker, 2017). It consists of a group of requests in which all of them must be successful. This mechanism ensures database consistency by coordinating multiple requests. If a single request fails, all the previous ones are rollbacked. For this reason, either all of the requests are successful or all fail, and the database remains consistent (Richards, 2015).

The microservices architecture is based on a distributed approach that suggests that each microservice has its database and communicates with the other services through messages. Figure 9 illustrates that when a service needs to send messages to multiple services, it is not possible to ensure an ACID transaction.

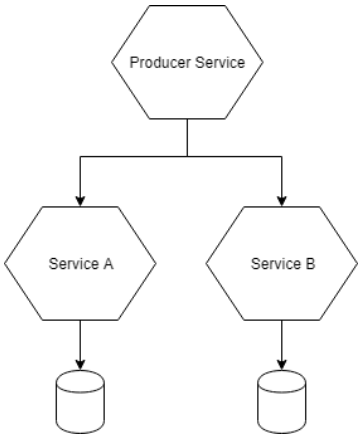


Figure 9 - Service transaction management.

For this reason, implementing business transactions that require modifications on multiple data sources is a challenge when using this architectural style as ACID transactions are applied to a single database (Ciavotta et al., 2017).

Instead of using ACID transactions, microservices usually follow the traditional service-oriented architectures approach and use BASE (Basic Availability, Soft state, Eventual Consistency) transactions. Therefore, the database will eventually be consistent but is not always consistent (Richards, 2015), which is a drawback of the microservices architecture that brings the advantage of greater availability of the system.

When the business transaction splits between different microservices, it is called a distributed transaction. More formally, a distributed transaction is “a transaction involving multiple transaction managers” (Gray and Reuter, 1993). This kind of transactions is difficult to manage and constitutes one of the main challenges of microservices architecture adoption, but it usually happens when splitting a monolithic system to a microservices architecture as previously local

ACID transactions that occurred in the monolithic system will now be executed across the microservices oriented distributed system (Cerny et al., 2017).

Thus, the problem addressed is the lack of knowledge and solutions to some of the challenges faced while adopting the microservices architecture and the difficulty of implementing distributed transactions, including data inconsistency or operations that were not entirely successful due to failed requests in the middle of a distributed transaction. The contributions of this work to those problems are detailed in the following sections of this chapter.

5.2 Objectives

The first goal of this work is to identify the current main challenges faced when adopting the microservices architecture and the solutions commonly used, providing insights regarding the architectural style for both researchers and industry practitioners.

Also, one specific challenge – management of distributed transactions – will be further detailed and possible solutions analysed. This work intends to determine if there are viable solutions to this issue in a microservices architecture and provide an implementation for easier management of distributed transactions using one of those solutions in a microservices context.

Therefore, research was conducted on microservices adoption processes, and issues reported, both in industry and in literature. Furthermore, existent solutions for distributed transactions and challenges while implementing the identified alternatives were analysed and a solution proposed.

Then, experts of the field also validated these results to ensure the viability of the solutions.

Summing up, this work has the following two main objectives:

1. Identify the most common technical challenges that teams currently face while migrating to the microservices architecture and possible solutions.
2. Address the distributed transactions challenge specifically, proposing a solution to ease the management of distributed transactions in a microservices architecture, using choreographed sagas.

5.3 Contributions of this work

The first main contribution of this work is the analysis and comparison of the available literature regarding microservices with industry practitioners' reports and observation. This work tries to clarify what are the current most common challenges while adopting microservices, guiding future researchers, and helping the industry to avoid the identified issues, while providing possible solutions and best practices also identified in the research. Also, the result of this analysis is compiled into a catalogue of activities or patterns related to architectural refactoring in the context of microservices.

Finally, this work also provides an implementation of the saga pattern, which intends to facilitate the management of distributed transactions in the microservices architecture, supporting teams with a solution to implement this pattern following a choreographed approach.

Furthermore, an article may be developed based on the output of this work and publicly published on a recognised platform or conference so that a different perspective of reviews can be gathered. Also, in the future, the developed solution can be applied in open source projects or put in practice by other interested companies which can then provide their testimony on the value of this project.

This document was written in English so that the final solution can reach a larger number of professionals or companies, thus contributing to the dissemination of results and potentially benefiting a variety of stakeholders.

5.4 Work methodology

The development of this work consisted of different phases:

- First, a narrative literature review was performed to validate the problem. With that information, the objectives of this work were defined, and therefore, this work aims to reduce the gap of missing research regarding microservices migration challenges by identifying the most commonly reported problems. Also, another objective is to support the management of distributed transactions in a microservices architecture by further detailing the issue and exploring possible solutions through a set of experiments. This first literature review helped define the context of the work and the problem to be solved. Also, the gathered information helped conclude the value analysis of this project.
- Once the problem context and the objectives of this work were defined, literature research was performed through a systematic mapping study that helped identify the most common challenges of microservices architecture adoption. Also, an industry questionnaire regarding the same topic was created to corroborate the findings of the systematic mapping study research. Finally, a participant observation study was also conducted in order to understand industry practices, challenges and solutions.
- Regarding the distributed transactions issue, a set of existent approaches were analysed, and the most adequate to the context of this work was chosen. A solution to implement that approach was then designed and implemented.
- Finally, the created solution and research findings were presented to the industry and evaluated through a survey in which expert professionals of the field can assess it and provide insights regarding the work developed. Furthermore, the solution was implemented on a microservices architecture to evaluate if it brings value and can solve the problem this work proposes to address.

6 Microservices migration research

After analysing and comparing existent research regarding microservices (see Section 4.1), the research plan for this work was defined. This chapter presents the research plan design, followed by its results and concludes with a summary of the findings, and threats to validity.

6.1 Design

6.1.1 Requirements

After analysing previous work regarding research on microservices migrations (Section 4.1), it was identified that there was not much research regarding the challenges found while migrating to a microservices oriented architecture. Most of the existent studies focus on architectural patterns, best practices or the current state of the architectural style to gather information for future research. Also, they analyse literature, and only a few gather data from the industry. The ones that examine data from the industry do not compare their results with literature findings. Furthermore, the research is still evolving, and there have not been many studies in recent years.

For these reasons, the following requirements were defined for this research (Table 9).

Table 9 – Microservices migration challenges requirements

Requirement number	Description
1	Focus on the challenges of microservices adoption
2	Identify the most common challenges
3	Compare data between literature and the industry
4	Only analyse data published since 2018

6.1.2 Design alternatives

To accomplish the defined requirements, alternative approaches can be used. To gather data from the literature, different methods of literature review can be applied, and they constitute design alternatives. From the industry side, there are also different techniques of data collection to be implemented. Below are described and analysed available options and possible alternatives that can be used in future work.

Literature research

To gather data from the literature, the following possible methods were identified.

1. Narrative literature review (NLR): this kind of literature review is also usually called traditional literature review. The reviewer gathers and interprets literature in a given field, without explicitly stating the inclusion and exclusion criteria used to select documents for the review. NLR follows a subjective approach and the rules applied to select the studies that are included in the analysis are usually based on the perspective of the reviewer. It is often used for 'opinion' pieces, 'expert' reviews or students' theses, but it is not useful to contribute to an informed debate. The reason for this is that the inclusion criteria may be 'biased', as mentioned above. Furthermore, as the search strategy is not clearly defined, it is not possible to replicate the review. Also, relevant studies could have been excluded as this kind of analysis does not follow a systematic and exhaustive approach. Finally, the quality of the studies included in this kind of review is usually not accessed. This means that low-quality studies may be included in the analysis while other studies of higher quality are excluded (Torgerson, 2003).
2. Systematic literature review (SLR): A systematic literature review follows a more rigorous approach when compared to the previously mentioned method. The search strategy used on an SLR is explicit and open to scrutiny. This research technique identifies all the available evidence regarding a specific theme. The data collected is then screened for quality and synthesised into an overall summary of the available research. As all found evidence is included in the study and the selection is catalogued specifying the reasons for inclusion or exclusion of specific evidence, the results are often less susceptible to subjectivity and can be replicated. This technique is useful as a way to "summarise the results of primary research and for checking consistency among such studies" (Torgerson, 2003).
3. Systematic Mapping Study (SMS): Systematic Mapping Studies are frequently seen as a simplified version of SLR and can, therefore, accomplish similar results in less time. The use of SMS in software engineering is considered to be very consistent and valuable in the last years. Like in an SLR, all the evidence of a specific field is analysed and screened following a systematic approach where all the criteria used are described. For this reason, the study can be easily replicated. Furthermore, this technique creates a map of the findings according to a previously defined classification framework, which gives a clear view of the results (Sampaio, 2015).

Industry data collection

Analysing the previous work, it is possible to conclude that there are few studies comparing literature research with industry professionals' experiences. Therefore, it is essential to collect quality data from the industry so that it can be compared against the data obtained from the literature research. With that objective, the following alternatives were analysed.

1. Questionnaire: The most common field method of industry data collection. It consists of sets of questions provided and answered in a written format. For this reason, they can be easily and quickly administered. To ensure valid results, the questions cannot be ambiguous, and the ordering and layout of the questionnaire must be carefully designed. Questionnaires are time and cost-effective, as they do not require to schedule any session to gather data. They can be answered when a software engineer has time between tasks. Furthermore, web-based questionnaires have no costs as the questions are delivered through the internet and data received in electronic form. Some of the disadvantages of this technique are ambiguous and poorly-worded questions, which can be problematic. Furthermore, even though it is relatively easy to answer a questionnaire, return rates may be low. To refute a null hypothesis, high response rates are needed. However, to understand trends with reasonable confidence, low response rates are acceptable. Another disadvantage of questionnaires is that responses may be highly subjective and based on the respondents own opinion (Lethbridge et al., 2005).
2. Interviews: Interviews consists of a face to face conversation between one researcher and one respondent. Similarly, as with questionnaire, before the interview, a fixed list of carefully worded questions must be defined. The difference to the questionnaire questions is that in an interview, the researcher can clarify any doubts that the respondent might have regarding the questions. Furthermore, the interviewer can deviate slightly from the script if he decided it is positive for the study. For this reason, one of the main advantages of this technique is that it is highly interactive and flexible. As for disadvantages, interviews are time and cost-inefficient. A meeting must be scheduled, which means both the researcher and the respondent must be available at the same time. Furthermore, one of the participants needs to travel to the defined location, usually the researcher. As stated in the questionnaire, responses may be subjective (Lethbridge et al., 2005).
3. Shadowing/Observation: The shadowing concept consists of the experimenter (the researcher) following a willing participant while recording its activities. These activities can include software engineers engaged in their work, or specific experiment-related tasks, such as meetings or programming. Observation is a similar technique in which the experimenter observes multiple willing participants instead of only one. The main advantage of this method is that it gives fast results and is easy to implement as long as there are willing participants, and the disadvantage is that the observer must have a good understanding of the environment so that he can understand what is happening and record what is essential for the study (Lethbridge et al., 2005).

4. Participant observation: This technique has some similarities with shadowing/observation. The difference is that in participant observation, the observer becomes part of the team it is observing and participates in key activities. The main advantage of this technique is that software engineers are comfortable with the researcher presence and act naturally, not influencing the study results. The disadvantage is that the researcher may become too involved and lose perspective of what is being observed (Lethbridge et al., 2005).

6.1.3 Final design

This study must be objective so that it brings the most value for this work. Therefore, regarding literature research, to facilitate the assessment of the validity of the findings, the reader should be able to replicate the study. For this reason, a systematic method must be used. As this work presents some time restrictions, it would not be possible to apply an SLR approach properly. Therefore, to accomplish the stated requirements, a systematic mapping study was used as the literature review method as it allows an objective and systematic review of the existent literature from a wide range of sources, in less time than an SLR. Furthermore, creating a map of the findings gives a clear view of the results. Finally, in the software engineering field, mapping studies are considered to be very consistent and valuable (Sampaio, 2015).

Regarding the industry data collection, a combination of the identified alternatives was used to make up for the disadvantages of each technique individually. The primary method used was the questionnaire, as it is the most cost-effective and commonly used. Furthermore, it does not require much time from respondent allowing the study to gather data from a broader range of professionals. However, it was clearly stated that the response rate is usually low in the software engineering field. For this reason, interviews should also be performed to experts of the area to gather additional valuable data. Finally, to avoid subjective information that can be provided while using questionnaires and interviews, shadowing/observation and participant observation was also performed so that a more accurate field perspective is included in the study.

The defined general approach is detailed in the diagram in Figure 10. Some tasks can be done in parallel to optimise time. Therefore, the work should begin with participant observation as it can be done right from the start, at the same time that a search for willing participants of shadowing/observation is performed. Furthermore, while performing these two activities, the literature review protocol and questionnaire can be designed. After designing these last two components of the study, they are put into practice by administering the questionnaire through different channels and starting the literature research at the same time.

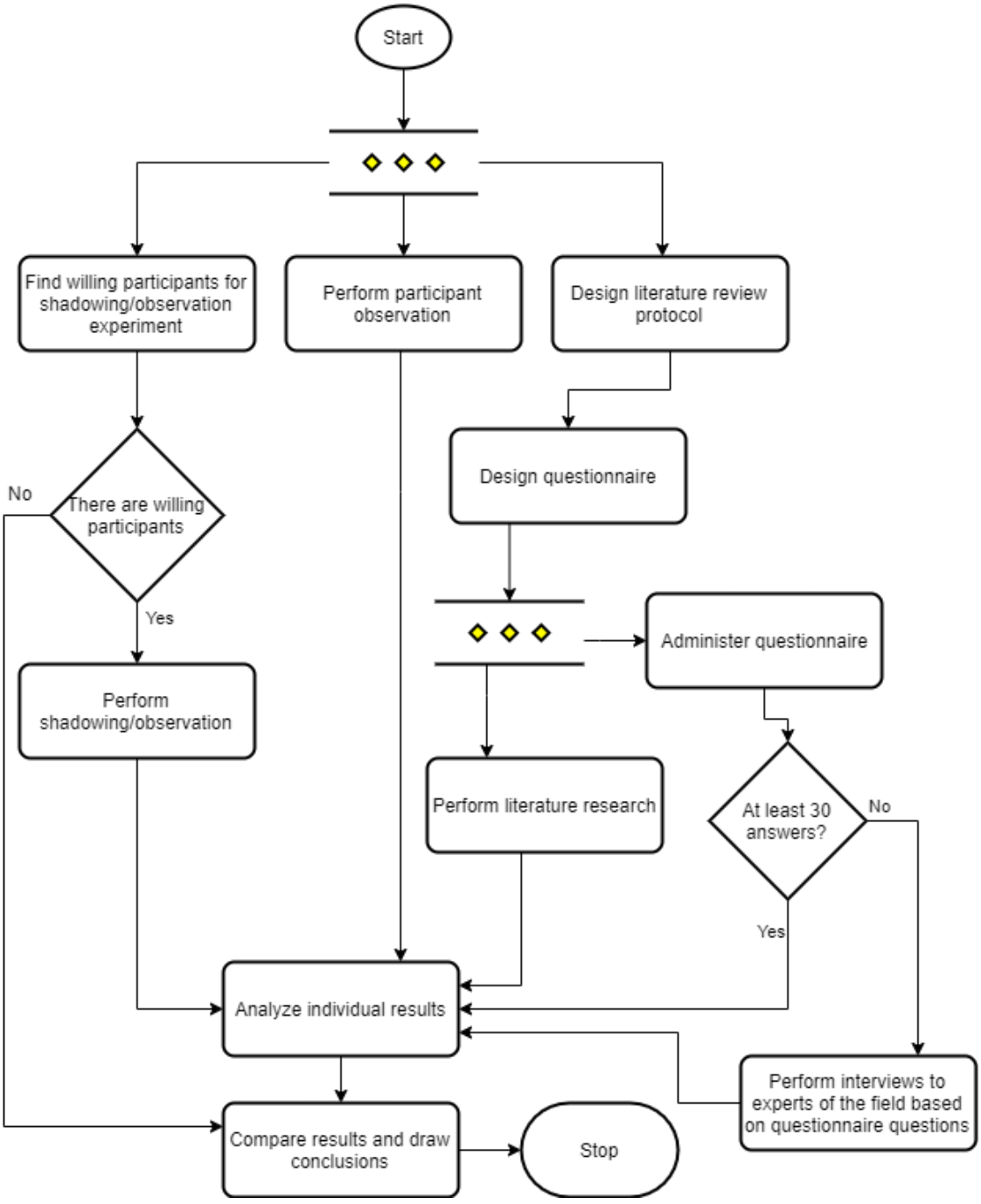


Figure 10 - Microservices migration challenges study design

In the case of having a low response rate on the questionnaire, interviews should be performed to gather more evidence from industry experts. When all the components of the study are finished, their results are analysed individually and then compared together to assemble the conclusions of the study.

On the following sections, the detailed design of the literature research and questionnaire are described.

6.1.3.1 Literature research design

The literature review described in the following sections presents a high-level overview of existent research on microservices migration challenges. Furthermore, this review has the objective of selecting and synthesising the reported experiences and challenges faced while migrating to the microservices architecture, appraising all high-quality evidence relevant to identifying the most common problems of the process.

The method used for this systematic mapping study is the one suggested by Sampaio and is based on guidelines provided by multiple mapping studies on the software engineering field, and by orientations for systematic reviews like the Cochrane Collaboration initiative (Sampaio, 2015).

Therefore, the study follows the six stages defined in Figure 11. These stages are as follows (Sampaio, 2015):

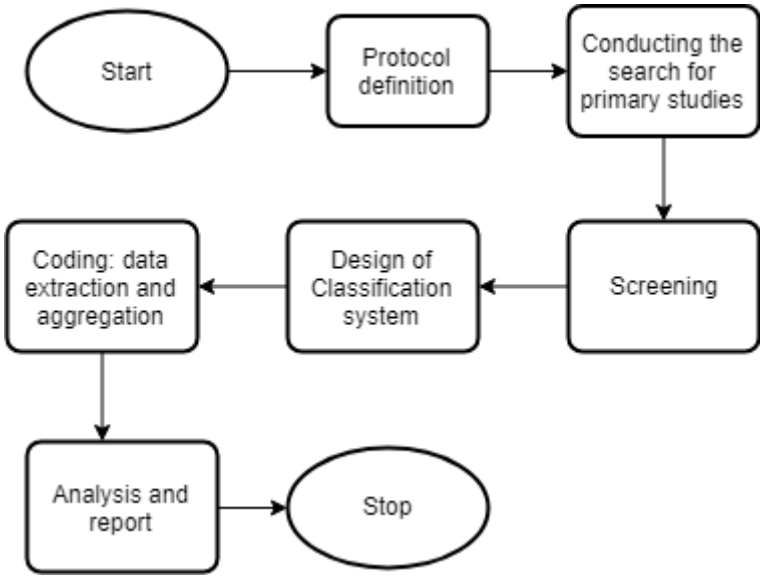


Figure 11 - Systematic mapping study stages

- The protocol definition stage consists of the design stage of the research and contributes to the reliability of the literature review and allows other researchers to analyse or replicate the study more clearly.
- The stage of conducting the search for primary studies consists of identifying the initial studies of the research. Every search information should be recorded. Then, possible duplicates should be removed and the primary studies identified.
- Then, the screening stage consists of applying the inclusion/exclusion criteria defined on the protocol to select only the valuable evidence. The used rules should be registered, and decisions recorded.
- After the screening stage, a classification framework should be designed. The primary objective of this artefact is to organise the selected papers in line with the research questions defined in the protocol or search strategy. If the identified area is too broad, research questions or inclusion/exclusion criteria should be refined in this stage.

- The coding stage consists of mapping the identified studies to the categories of the classification framework and extracting and recording data to answer research questions.
- Finally, the last stage consists of analysing the gathered data and reporting the findings. On this stage, the research questions are answered, and threats to validity described.

Therefore, as the protocol is the design stage of a mapping study, the rest of this section describes the defined protocol and search strategy.

Review boundaries

With a scoping performed before the mapping study, it is possible to identify that there are multiple kinds of literature regarding the microservices architecture and migration processes. The key material for this study are reports of performed migration processes, industry surveys, and interviews regarding the topic, studies of common architectural and design patterns used on this kind of systems, and other similar systematic literature reviews. Only literature published since 2018 was evaluated to assure only cutting-edge knowledge was considered. This study will use only digital libraries as they are the most used repository of research items on the software engineering field (Sampaio, 2015), namely Google Scholar, IEEE Xplore, and ACM Digital Library.

Research questions and goals

The primary goal of this review is to identify and analyse the most common challenges reported during microservices migrations and possible solutions used to solve them. Also, it can help researchers, students and industry practitioners better understand the available knowledge on the microservices subject and existent migration processes.

The PICOC (Population, Intervention, Comparison, Outcomes, Context) model is a commonly used framework on software engineering research to frame research questions to deliver well-focused research (Sampaio, 2015). Using this model, the described target and goals of the review can derive into the research questions presented in Table 10, Table 11, and Table 12.

Table 10 - Research question 1 following the PICOC framing (RQ_1)

Research question	What are the most common problems when migrating a monolithic system to the microservice architecture?
Population	Reports of performed migration processes, industry surveys, and interviews regarding the topic
Intervention	Identifying the key attention points while performing a microservices migration
Comparison	Number of times the challenge is referred on the reviewed literature
Outcomes	The reported challenges with most references on the reviewed studies
Context	Research on microservices migrations, and reported experiences in the industry
Rationale	There is a multitude of documents of different types and from various sources and contexts reporting challenges with microservices architecture and migration. RQ_1 intends to map the available knowledge, identifying the most common challenges reported across the reviewed literature.

Table 11 - Research question 2 following the PICOC framing (RQ_2)

Research question	Which of the problems reported are avoidable?
Population	Reports of performed migration processes detailing problems identified. Also, industry surveys, and interviews that report issues on this kind of migration.
Intervention	Classifying identified problems as avoidable or intrinsic
Comparison	Challenges are reported on some migrations but avoided on others
Outcomes	Identification of avoidable problems of this kind of processes. Identification of intrinsic problems to the microservices architecture adoption or migration
Context	The same as RQ_1
Rationale	The microservices architecture some key attention points. The objective of this research question is to distinguish the avoidable problems and the intrinsic problems that occur when refactoring software systems to the microservices architecture.

Table 12 - Research question 3 following the PICOC framing (RQ_3)

Research question	What are the most common solutions to adopt microservices architecture?
Population	The same population of RQ1 and studies of common architectural and design patterns used on this kind of systems, and other similar systematic literature reviews.
Intervention	Identifying similarities of practices and approaches while performing architectural refactors to the microservices style.
Comparison	The number of times a specific strategy or technique is used successfully in migrations.
Outcomes	A set of successful patterns to be applied to this kind of migration processes.
Context	The same as RQ1.
Rationale	Research on the adoption of microservices architecture is evolving. However, there is still "a lack of systematic guidance on the refactoring process for existing monolithic applications" (Fritzsch et al., 2018). This research question has the objective of identifying successful patterns used in reported migrations and experiences.

Screening method and Inclusion/Exclusion (I/E) criteria

After defining the boundaries of the review, its goals, and research questions, the selection criteria were derived from the framed RQ to guide the screening process of the systematic mapping study. During the screening process, the title, keywords, and abstracts of the primary studies should be analysed. When these selected parts are not enough to decide following I/E criteria, further parts of the document should be analysed, mostly the conclusions.

The defined criterions are described in Table 13.

Table 13 - Microservices migration challenges systematic mapping study applied I/E criteria

Criterion	Description
I1	Technical reports of performed microservices migrations.
I2	Studies describing the problems faced with the microservices architecture or during migration to this architectural style.
I3	Industry surveys and interviews regarding experiences while doing microservices migrations.
I4	Studies providing architectural solutions, methods or techniques (e.g., tactics, patterns, styles, views, models, reference architecture) specific for the microservices architecture or distributed systems in general.
E1	Studies published before 2018.
E2	Studies not available as full-text.
E3	Books.
E4	Studies where microservices are only used as an example.
E5	Studies not written in English.

Search string

Finally, to conclude the search strategy applied in this mapping study, the search string is defined aligned with the previously described I/E criteria and framed research questions:

$\big((microser * OR\ micro - servi * OR$ "micro servi" $*)\ AND\ (refact * OR\ reengin * OR\ migrat *)\big)$

6.1.3.2 Questionnaire design

To collect data from industry experiences, a questionnaire must be administered to industry professionals. Since the questionnaire targets professionals of the software industry, the respondents did not have difficulties to answer a web-based questionnaire. As this type of questionnaire also lows the cost of the data collection, it was the one selected.

Appropriate design is essential to gather quality data and valid responses. Therefore, the questionnaire was designed based on some practical guidelines recommended by Saaya et al. in 2007.

First of all, the questionnaire should motivate the respondent to answer since the beginning. For this reason, a welcome message introducing the respondent to the questionnaire context is mandatory (Saaya et al., 2007). Then, some screening questions should be included to verify if the respondent belongs to the research population. Regarding the layout of the questionnaire, it depends on the content and the number of questions of the questionnaire. Ideally, it should be a non-scrollable single page. However, if that is not enough, it must be divided into pages that fit within a monitor screen. This can cause too many pages if there are many questions. Therefore, in that scenario, the questions should be divided into sections of the subject being studied (Saaya et al., 2007).

The questions can be close-ended (dichotomous questions, multiple-choice, rank order scaling, rating scale, constant sum) or open-ended (free text). However, there should always be an option to provide an open-ended answer if the respondent wants to, which ensures that the respondent can provide all the information he finds necessary and is not limited by the provided answers style (Lethbridge et al., 2005).

At the end of the questionnaire, an appreciation message should be present. On this section, the respondent should be able to contact the questionnaire author or provide any additional comments (Saaya et al., 2007). Following the described guidelines, Figure 12 presents the recommended structure for web-based questionnaires.

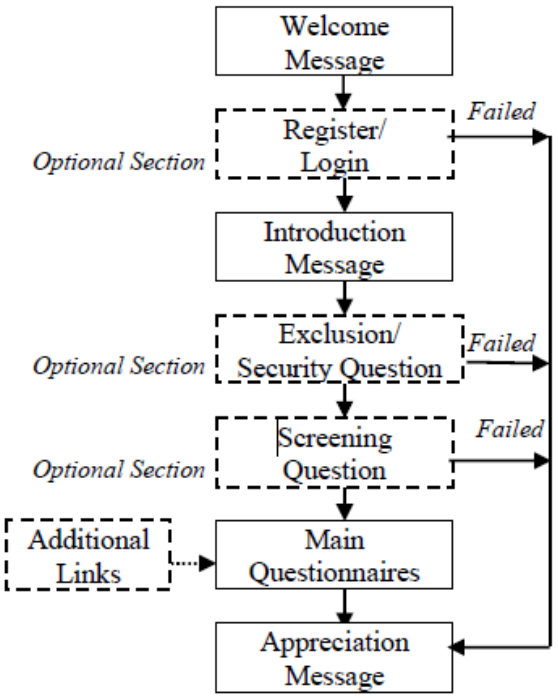


Figure 12 - Questionnaire overall structure (Saaya et al., 2007)

There are available tools that automatically deal with the framing, forms, and fields of the questionnaire and its structure. It also deals with usability and navigation in the questionnaire and provides multiple types of response formats and question types.

Furthermore, it provides an easy to analyse answer set by presenting some graphics illustrating the answers statistics. Google Forms was the tool used in this work as it provides all these features while following the recommended guidelines used for the questionnaire (Google, 2019). Furthermore, it allows the questionnaire to be answered only once by each respondent using the google login but remaining anonymous. The login functionality is the "Exclusion/Security question" mentioned in Figure 12, as it only allows each respondent to answer once.

Regarding the layout of the questions, following the recommendations described above, they were grouped into different sections according to the topic. The chosen sections were:

1. Cover page: This section presents a welcome message along with a description of the questionnaire's context. It also introduces the questionnaire structure and time expected to complete it. To achieve a snowballing effect, the respondent is asked to share the questionnaire with his contact network.
2. Introduction: This section has the purpose of gathering some information about the respondent and the migration experience he has. The answers to this section provide data regarding the years of professional experience of the respondent, his professional role, the stage of the migration that he is at the time of the answers. Furthermore, it contains some questions regarding the system before the migration and the reasons

that motivated the migration. It is composed of different multiple-choice questions and some text input where the user is only allowed to insert numbers (number of years of experience, for instance).

3. Existing system analysis: This section gathers data regarding the approach used to analyse the current system before the migration. The respondent is asked what sources he used to examine the system, the reasons to do it, and the four most significant challenges that he faced while analysing the existing system.
4. Designing the new architecture: The objective of this section is to understand the process used to design the new system. In this section, the respondent provides information about the activities performed while designing the new system and how it was documented. The respondent also describes if new functionalities were implemented during the migration or if only existent features were present in the new system. Then, the respondent indicates the four main challenges faced while designing the new system.
5. Implementing the new system: On this section of the questionnaire, some answers are provided regarding the process of microservices implementation. The respondent describes how he started the implementation, how the first functionalities to migrate were selected, what was the adoption process used and how data migrations were performed if there were data model or database modifications. Finally, a final question asks which the four main challenges were faced while implementing the new system.
6. Additional feedback: This final section contains three optional open-ended questions, one allowing the respondent to provide any additional information regarding his experience while migrating a system to a microservices architecture. Then, he is asked to provide feedback regarding the questionnaire, and he can subscribe to the study results by providing his e-mail address. An appreciation message ends the questionnaire by thanking the respondent for his contribution.

Most of the questions provided are multiple-choice, but at the end of each section, the respondent can answer an optional open-ended question to give any additional information. The options for multiple-choice questions were obtained from multiple industry reports regarding experiences of microservices migration. All the multiple-choice questions provide an alternative “Other” where the respondent can specify any answer not present in the options provided. The complete questionnaire is present in the appendix A.

After designing the questionnaire, it was administered to a closed group of engineers to evaluate its quality before widely distributing it. The provided feedback helped improve the wording of some questions to avoid ambiguity. Furthermore, the division between the sections was more clearly defined so that respondents can clearly understand if they are answering to the design or implementation phase. Also, two orthographic mistakes were detected and fixed.

The questionnaire will be distributed through mailing lists to industry professionals from different companies, published in forums of communities with interest in the field and also shared in networks like LinkedIn.

6.2 Data from research literature

6.2.1 Conducting the search for primary studies

Defined the protocol of this review, the search string was applied on the selected digital libraries in order to gather the primary studies. Only studies published since 2018 with full-text available in pdf format were considered. The search on Google Scholar identified 15 documents, IEEE Xplore found 19 documents, and ACM Digital Library obtained 31 documents, forming a total of 65 results. The search was performed on the 1st of February 2019. After merging the three result sets and removing duplicates, 54 documents were left.

6.2.2 Screening

After gathering the 54 primary studies, the selection criteria (see Table 13 at Section 6.1.3.1) were applied, and the following 18 documents were identified.

Table 14 - Selected papers after the screening stage of the systematic mapping study

Article Identifier	Document Title	Author(s)	Publication Year
1	A pattern language for scalable microservices-based systems	Gastón Márquez, Mónica M. Villegas, Hernán Astudillo	2018
2	AjiL: Enabling Model-driven Microservice Development	Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf.	2018
3	From Monolithic to Microservices An Experience Report from the Banking Domain	Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, Manuel Mazzara	2018
4	An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions	Welder Luz, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto, Rodrigo Bonifácio	2018
5	Architectural Patterns for Microservices: A Systematic Mapping Study	Davide Taibi, Claus Pahl, Valentina Lenarduzzi	2018
6	Migrating towards Microservice Architectures: an Industrial Survey	Paolo Di Francesco, Patricia Lago, Ivano Malavolta	2018

Article Identifier	Document Title	Author(s)	Publication Year
7	Interface Quality Patterns — Communicating and Improving the Quality of Microservices APIs	Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, Cesare Pautasso	2018
8	Limiting Technical Debt with Maintainability Assurance – An Industry Survey on Used Techniques and Differences with Service- and Microservice-Based Systems	Justus Bogner, Jonas Fritzsch, Stefan Wagner, Alfred Zimmermann	2018
9	Microservices	Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert	2018
10	Migrating Web Applications from Monolithic Structure to Microservices Architecture	Zhongshan Ren, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei, Tao Huang	2018
11	Migrating towards Microservices: Migration and Architecture Smells	Andrés Carrasco, Brent van Bladel, Serge Demeyer	2018
12	Migrating Enterprise Legacy Source Code to Microservices On Multitenancy, Statefulness, and Data Consistency	Andrei Furda, Colin Fidge, Olaf Zimmermann, Wayne Kelly and Alistair Barros	2018
13	Partitioning Microservices: A Domain Engineering Approach	Munezero Immaculée Josélyne, Doreen Tuheirwe-Mukasa, Benjamin Kanagwa, Joseph Balikuddembe	2018
14	Query Strategies on Polyglot Persistence in Microservices	Luís H. N. Villaça, Leonardo G. Azevedo, Fernanda Baião	2018
15	Strategy and procedures for Migration to the Cloud Computing	Naim Ahmad, Quadri Noorulhasan Naveed, Najmul Hoda	2018

Article Identifier	Document Title	Author(s)	Publication Year
16	Towards Micro Service Architecture Recovery: An Empirical Study	Nuha Alshuqayran, Nour Ali, Roger Evans	2018
17	Using Microservices for Legacy Software Modernization	Holger Knoche and Wilhelm Hasselbring	2018
18	Microservices migration patterns	Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, Theo Lynn	2018

6.2.3 Classification system

After the papers have been screened according to I/E criteria, it is essential to develop a classification system that enables the aggregation of the identified papers and the extraction of data from the generated groups of documents. Therefore, considering the previously defined research questions, the classification framework of Figure 13 was developed.

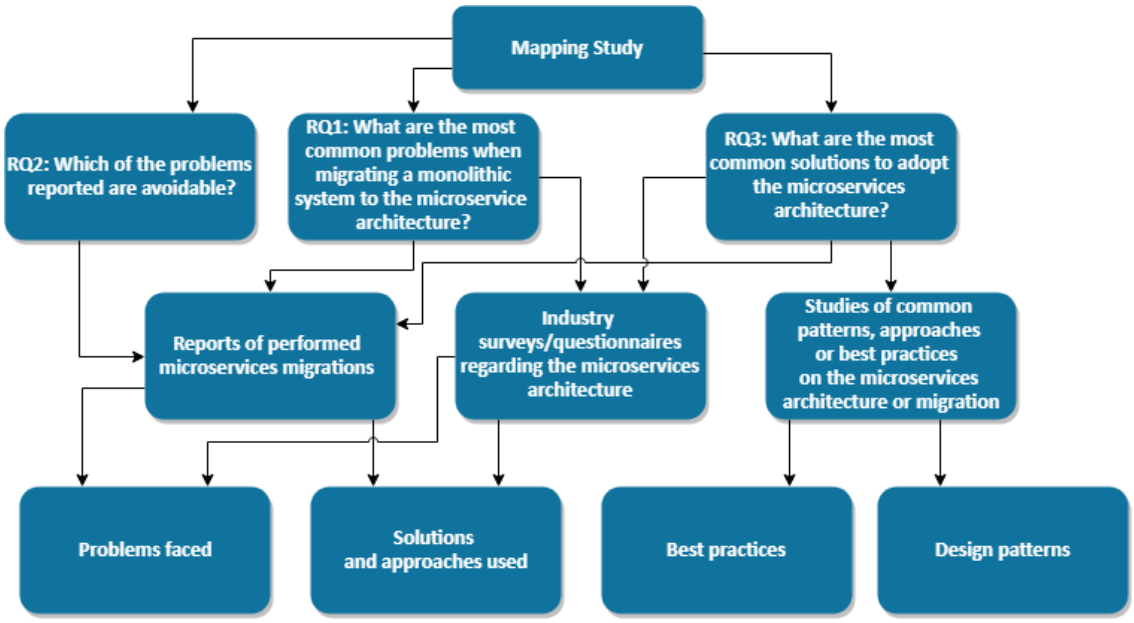


Figure 13 - Systematic mapping study classification framework

6.2.4 Coding: data extraction and aggregation

Framed the research questions, selected the papers respecting the defined I/E criteria and developed the classification framework accordingly, the data of selected studies were extracted and aggregated in the respective categories of the classification framework (see Figure 13).

From this analysis, the following information can be identified:

Table 15 contains the problems identified,

- Table 16 the solutions and approaches used,
- Table 17 shows the best practices, and
- Table 18 the design patterns mentioned in the analysed literature.

Table 15 - Problems identified in systematic mapping study

Description	**References (Article Identifier)**	**Number of references**
Testing Complexity on the microservices architecture	5, 6, 9	3
Implementation Effort and Infrastructure Management is more laborious than in monoliths (especially at the beginning of the development)	5, 6	2
Network related issues	5, 11, 17	3
Data consistency issues	9, 11, 12, 17	4
Distributed monitoring and logging	6, 9	2
Decomposition of the pre-existing system with the proper granularity and low coupling	5, 6, 9, 11, 17	5
Create uniformity across services	6	1
Continuous deployment and monitoring	5, 9, 11	3
State management	9, 12	2
Security	5, 9	2
Database splitting	9, 10	2
Inexperienced Team	9, 11	2

Table 16- Solution and approaches identified in systematic mapping study

Description	**References (Article Identifier)**	**Number of references**
Identification of the dependencies of the pre-existing system modules	6, 9, 15, 17, 18	5
Careful design of the business workflows	6, 15, 18	3
Domain Driven Design	3, 6, 11, 13, 14, 17, 18	7
Phased adoption	5, 6, 11	3
Parallel adoption	6, 11	2
Data kept ‘as is’ in the pre-existing system	6	1
Data was migrated to the new system by implementing data migration procedures	6	1
Static code analysis to identify modules to be decomposed both in services and databases	10, 17	2
Generation of microservices through Model-driven engineering	2, 16	2

Table 17 - Best practices identified in systematic mapping study

Description	References (Article Identifier)	Number of references
Microservices implementing self-healing	5	1
Prioritise the components to be migrated through a previously defined criterion (amount of dependencies, amount of users of specific functionality, etc.)	6, 9, 18	3
Use DevOps methodology to automate testing and deploy stages applying Continuous Integration and Delivery (define the DevOps strategy before starting the migration and have a separate autonomous DevOps pipeline for each microservice)	9, 11, 14, 15, 17, 18	6
Adopt an Agile approach	9	1
Monolith First: Start with a monolith and split it into microservices incrementally through refactoring instead of completely redeveloping the whole system into microservices	9, 11, 18	3
Messages between microservices should be encrypted and authenticated	9	1
Have at least some experienced developers (in distributed systems or microservices) in each development team	11	1
Provide fine-grained, well defined and documented microservices APIs and Interfaces	11, 14, 17, 18	4

Table 18 - Design patterns identified in systematic mapping study

Description	References (Article Identifier)	Number of references
Service discovery	5, 1	2
Lightweight Containerization (Ideally a container per Service)	1, 11, 14, 15, 16, 18	6
Database per service	5, 1, 10	3
API Gateway and Strangler Pattern	5, 17, 18	3
Stateful messaging pattern	12	1
Partial state deferral pattern	12	1
State repository pattern	12	1
Stateful service pattern	12	1
Command Query Responsibility Segregation (CQRS)	14	1
Event Data Pump	14	1
Circuit Breaker	1, 16, 17, 18	4
Load Balancer	1, 16, 18	3
Configuration Server	18	1

6.2.5 Analysis and report

At this point in the study, all the required information was extracted to the tables presented in Section 6.2.4. This section analyses the results of the study and draws conclusions regarding the framed research questions.

6.2.5.1 Research Question 1 (RQ_1)

Regarding RQ_1 ("What are the most common problems when migrating a monolithic system to the microservice architecture?"), the five most referenced problems in literature are presented in Table 19. Thus, these issues are reported with some consistency in the process of microservices adoption.

Table 19 -Five most referenced challenges in the literature

Number	Name	Number of references
1	Decomposition of the pre-existing system with the proper granularity and low coupling	5
2	Data consistency issues	4
3	Network related issues	3
4	Testing Complexity on the Microservices architecture	3
5	Continuous deployment and monitoring	3

Regarding the first issue, the most mentioned challenge of microservices migration is defining the proper granularity for each service in order to ensure low coupling and high cohesion across the system. Teams find difficulties decomposing the existing system to the microservices architecture and defining the responsibilities of each one of the new services.

When the defined granularity is too coarse, the benefits of microservices are not worth it, and the teams end up with a distributed monolith instead of a microservices architecture, as the modules are not independently deployable and are highly coupled. This situation has all the disadvantages of the monolith that the team was trying to solve as the system remains with low maintainability, but now there is also the need to deal with distributed systems challenges.

However, when the granularity is too fine there are also some issues with performance due to network latency (Carrasco et al., 2018). For this reason, it is important to decompose the existing system following Domain-Driven Design strategies, such as the Business Capability pattern, where the system is divided by the multiple business capabilities or entities defined in the domain model.

Another highly referenced problem was "Data Consistency Issues". The analysed studies refer that while implementing Microservices, teams usually forget about the CAP theorem and deal with difficulties regarding Data Consistency (Carrasco et al., 2018). This problem refers to the

difficulty of keeping a consistent state across all the Microservices. In a monolith, there is usually a single database, which ensures data consistency as it is ACID and sequential.

However, in a distributed system such as the Microservices architecture, it is harder to ensure consistency as the information must be transferred between multiple services and databases. For this reason, to respect the CAP theorem, Microservices often adopt an eventual consistency approach, which means that the system only guarantees that if no new updates are made to the object, eventually all the databases will be consistent (Furda et al., 2018). The migration to the Microservices architecture must be planned considering data consistency across the distributed system in order to avoid unexpected data inconsistencies (Knoche and Hasselbring, 2018).

The last three most referenced issues had three references each:

- Network related issues due to the distributed nature of the microservices architecture are caused by the need for the microservices to communicate among them, which was not a need in the monolithic architecture. This communication increases the latency of the requests, and the system must also be able to deal with communication failures between the microservices in order to remain resilient. To achieve this, patterns such as the circuit breaker can be used.
- Testing complexity on the microservices architecture was another identified challenge as the integration between the microservices must also be tested, something that was not a requirement in the monolith as it was a single service and did not communicate with other components. One way to deal with this is to define Bounded Contexts as specified by Domain Driven Design and implement integration tests only between the components of those bounded contexts. This simplifies the boundaries of the tests, making them more maintainable. To test the system between different bounded-contexts, consumer-driven contracts (Chen, 2018) can be used.
- Finally, the last issue of the five most referenced was continuous deployment and monitoring. Each microservice should have a smaller codebase than the monolith, making the build and deploy processes faster. However, there are multiple services to be deployed in a microservices architecture. Therefore, if the services are not independently deployable, with individual continuous integration environments, the deployment process becomes highly complex. Furthermore, instead of monitoring a single service, in the microservices architecture, the different components are working together towards a common end, and should, therefore, be monitored together in order to easily visualise what is happening in the system and identify an issue when it happens. A technique to help with this challenge is log aggregation and correlation ids.

There are possible solutions to the problems identified in Table 19, and some of them were briefly mentioned. They may be able to completely avoid the problems or at least reduce their impact, depending on the kind of issue.

6.2.5.2 Research Question 2 (RQ_2)

RQ_2 intends to classify the issues as avoidable or intrinsic: "Which of the problems reported are avoidable?". Therefore, analysing the gathered documents, it is possible to classify the issues as described in Table 20. Challenges 1, 3, 4, and 5 are intrinsic to the distributed nature of the microservices architecture and even though they can be reduced and managed they cannot be wholly avoided. However, challenge 2 can be avoided entirely using the best practices of software engineering and applying Domain Driven Design techniques to define the new system boundaries.

Table 20 - Most common challenges classsification (avoidable or intrinsic)

Number	Problem	Classification
1	Data Consistency Issues	Intrinsic
2	Decomposition of the pre-existing system with the proper granularity and low coupling	Avoidable
3	Network related issues	Intrinsic
4	Testing Complexity on the microservices architecture	Intrinsic
5	Continuous deployment and monitoring	Intrinsic

6.2.5.3 Research Question 3 (RQ_3)

Regarding RQ_3 "What are the most common solutions to adopt the microservices architecture?", the systematic mapping study concludes that the five most used solutions are the ones described in Table 21.

Table 21 - Most common solutions to adopt the microservices architecture

Number	Solution	Type	Number of references
1	Domain-Driven Design	Approach	7
2	DevOps practices	Best Practice	6
3	Containerization/Container per Service	Design Pattern	6
4	Circuit Breaker	Design Pattern	4
5	Provide fine-grained, well defined and documented microservices APIs and interfaces.	Best Practice	4

The first, Domain-Driven Design, is an approach used to design the system modelled around the business domain, providing better maintainability and flexibility to business changes. It helps to identify well-defined boundaries of the system, achieving low coupling and high cohesion between the different components. Furthermore, when using Domain Driven Design, a ubiquitous language is created which facilitates the communication between business domain experts, stakeholders and software engineers. This approach helps to define the boundaries and granularity of the microservices when trying to decompose an existent monolith into the microservices architectural style, avoiding the problem 2 of Table 20.

Then, DevOps is a set of best practices to increase an organisation's ability to deliver software faster, which is also one of the benefits mostly referenced regarding microservices architecture.

Therefore, both concepts should be developed together to achieve this purpose. Some authors state that microservices are one of the practices of DevOps methodology and that the DevOps strategy should be defined before starting the migration to the microservices architecture, and each microservice should have a separate autonomous DevOps pipeline. These pipelines should include the build, test, and deploy stages of the software development lifecycle and should be highly automated, applying the concepts of Continuous Integration and Delivery. Monitoring and logging practices are also enforced by this methodology. Therefore, applying the DevOps methodology to microservices software development will largely reduce the intrinsic challenges 4 and 5 mentioned in Table 20.

While DevOps is all about quickly delivering software with high quality, automating the process along the way, containerization purpose is to package the software that is being deployed in an isolated way and optimizing infrastructural costs, therefore it is natural that Containerization is the third most referenced solution in the literature. Traditionally, software was developed to run in a specific environment and if the environment was modified, it could cause errors. Containerization main objective is to bundle the software with all the necessary configuration files, libraries and dependencies that are required to run the application in an efficient and bug-free way across different computing environments.

Furthermore, containerization is a virtualization technique and therefore multiple software containers can be run for a microservices architecture in one or more physical machines, where each microservice can run independently on its own container, optimising the resources of the physical machines that are running the containers.

The last two most referenced solutions both had 4 references:

- The circuit breaker design pattern wraps a specific remote call in a circuit breaker object, which keeps track of the failures of the call being monitored. If these failures reach a defined threshold, the circuit breaker trips and all further execution of the wrapped remote call will return an error without actually executing the call, protecting any further components that may be failing to respond. The circuit breaker can be reset automatically after a defined time interval, or by manual intervention, depending on the implementation. Often this pattern will protect against a range of errors that the protected call could raise, such as network connection failures, helping to reduce the intrinsic challenge 3 of Table 20.
- Finally, it was considered a best practice by multiple authors to have well-defined API and contracts in the implemented microservices. This interface should be fine-grained and documented. The reason for this is to ensure low coupling among the microservices and high flexibility for future changes. Also, by defining fine-grained interfaces the possibility of having breaking changes in the future is reduced. The documentation is essential for all the teams working in the microservices oriented system being aware of the functionalities provided by all the components and how to use them.

6.3 Data from industry

To gather data from the industry and to be able to compare it with data from the analysed literature (see Section 6.2), a questionnaire was distributed to multiple software industry professionals that have participated or are currently participating in the migration of a monolithic system to the microservices architecture. The questionnaire was shared online using LinkedIn, Twitter, and professional forums such as DevOps Porto community, and Agile Connect Porto. Also, it was presented and distributed in technological conferences (such as TechInPorto) and companies (such as Farfetch). Finally, it was distributed directly to experienced software engineers and software architects. In total, thirty answers were obtained.

6.3.1 Introduction

The first section of the questionnaire – Introduction - has the purpose of introducing the participant to the study and identifying the characteristics of the population being studied, and also discarding participants with no value for the study – for example, someone who is not experienced with microservices. Therefore, this section analyses the results obtained in this first section.

As can be analysed in Figure 14, the average years of experience of the participants are around 9 years. Therefore, we can consider the audience is composed of senior professionals, with only 7 participants having less than 4 years of experience in a total of 30 response submissions.

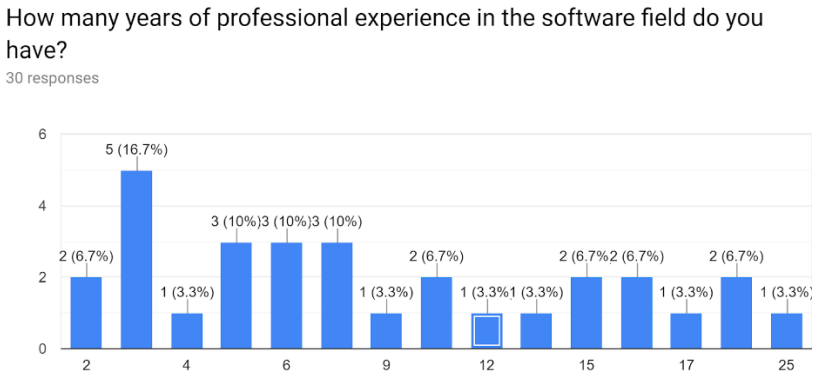


Figure 14 - Questionnaire - Participants professional experience (X: years of experience, Y: Number of responses)

Regarding the professional role performed during the migration, the participants were mostly Software Engineers (63.3%), and Software architects (30%), with only one Quality Assurance Engineer and one Project Manager, as can been in Figure 15.

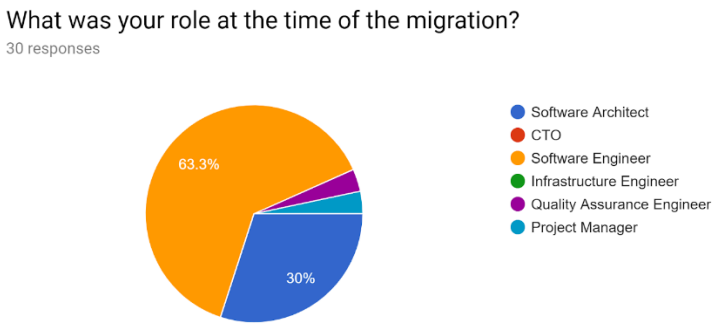


Figure 15 –Questionnaire - Participants professional role

The context of the migration performed by the participants was also investigated, to analyse if it can bring value for the study. As described in Figure 16, most of the systems consisted of monolithic applications, and only 10% of the participants stated that they migrated systems where only some of the services were monoliths. One participant stated that he performed a migration of a system with more than 100 services, which is a clear outlier.

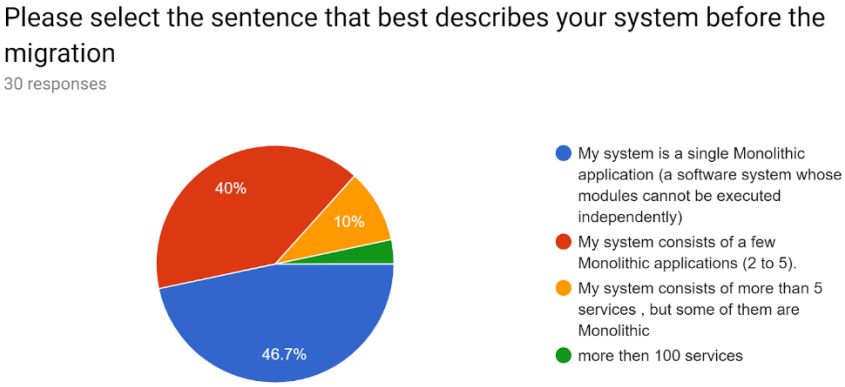


Figure 16 Questionnaire – Description of the system before migration

To better clarify the context of the migration, the participants were asked to identify the exact number of existent services before the migration. The average result was 16 services before the migration. However, removing the previously identified outlier which stated that the system had 330 services before the migration, the average becomes 5. This average is caused by multiple participants stating that their system had more than ten services, which highly increases the average result. However, most of the participants answered that their system only had between one and three services, with the highest concentration of participants (10) stating that they only had one service before the migration. Also, two participants had no services before, which means that they developed the microservices architecture from scratch without migrating an existing system, which can be better observed in Figure 17.

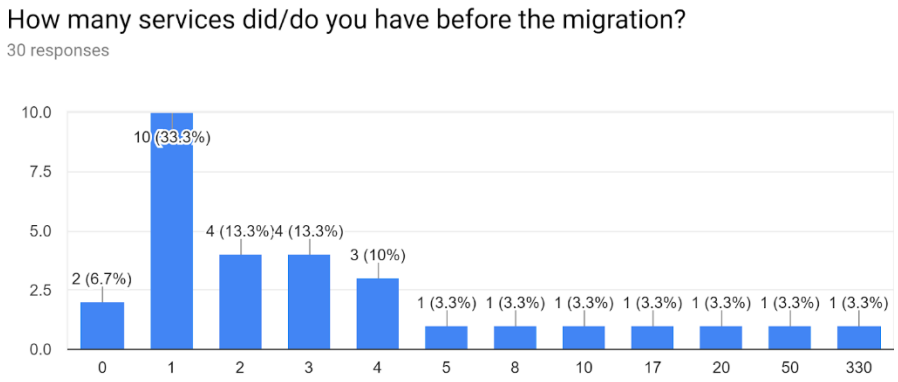


Figure 17 - Questionnaire - Number of services before migration (Y: Number of responses, X: Number of services)

Most of the participants have fully completed the migration or are close to completing, as can be observed in Figure 18.

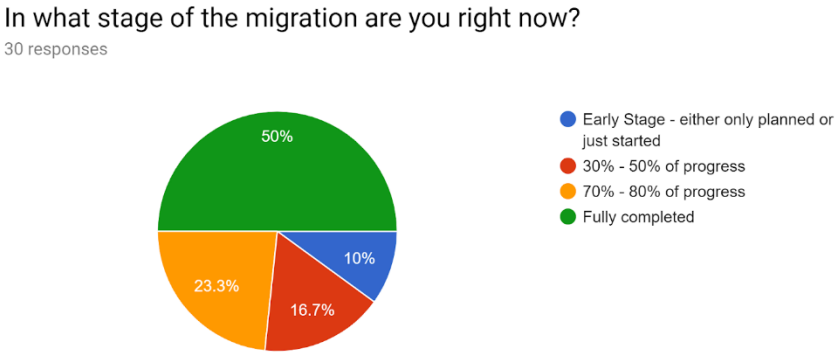


Figure 18 - Current stage of migration

The participants were also asked to compare the initial estimative of months necessary to complete the migration and the time that it really took. Therefore, the answers of the participants who already completed the migration (50%) were analysed, as only after completing the migration we can have the real time it took. Comparing the answers of these participants, we can conclude that most of the migrations got delayed, 28.6% of them took twice the time that was estimated. Only 21.4% were on time, which indicates that some unexpected challenges were faced during the migration, and therefore demonstrates the value of this study to try to identify these challenges and their solutions, better preparing professionals for their future migrations, which can be observed in Figure 19. It is also important to mention that most of the migrations took more than six months to complete.

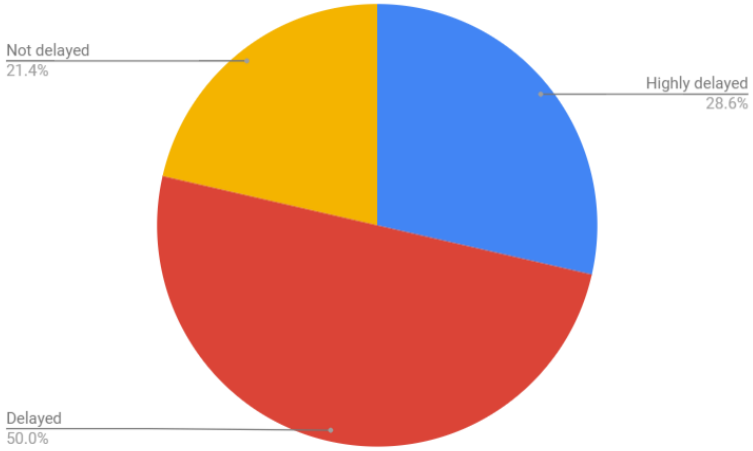


Figure 19 - Pie chart of migrations delivery time

To finish the first section of the questionnaire, participants were asked the four mains reasons that made them decide to migrate to the microservices architecture. The answers are present in Figure 20.

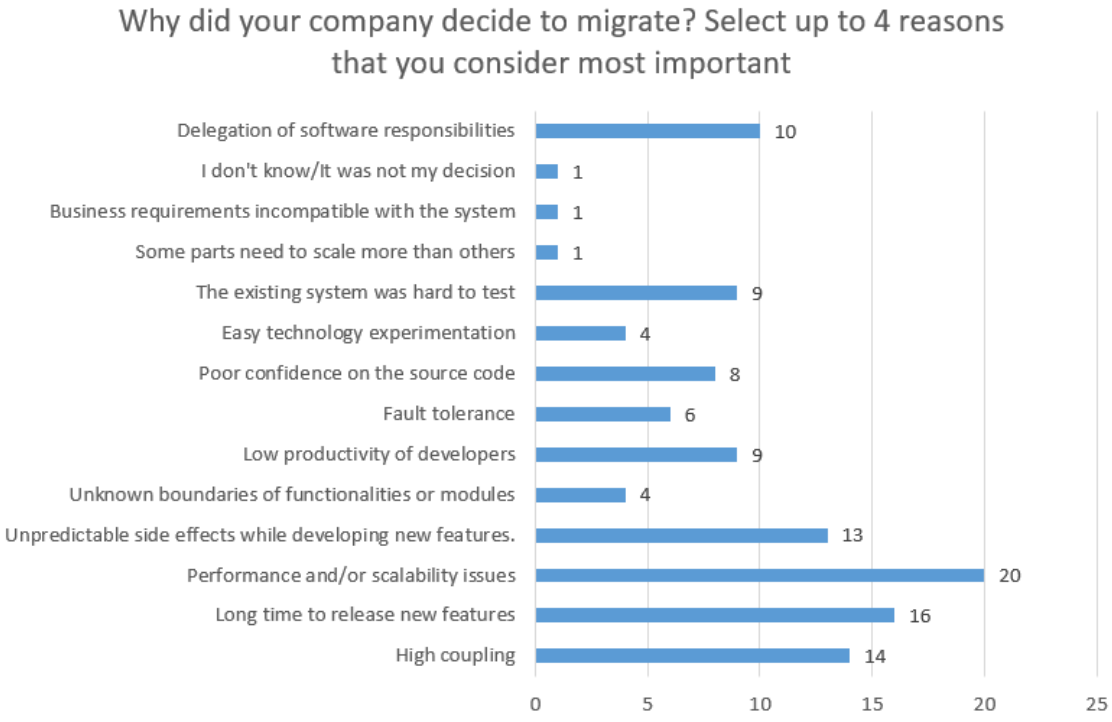


Figure 20 - Questionnaire - Reasons to migrate to microservices

Analysing Figure 20 it is possible to conclude that the most common reason for the migration was “Performance or scalability” issues with 20 participants indicating this option. Microservices allow more efficient management of the available resources, scaling better with increasing volumes of data. As previously mentioned, one of the main benefits of microservices is its flexibility to business changes and the possibility to quickly deliver new features. This is

mostly because of the low coupling of the microservices architecture, making it possible to modify a part of the system without affecting any other module - this statements are supported by the second and third most indicated reasons to migrate by the industry participants: “Long time to release new features” and “High coupling” (of the monolith).

6.3.2 Existing system analysis

This section presents the answers to the second section of the questionnaire, which is focused on the analysis of the existing system before the migration.

First, the participants were asked to specify the sources they used to analyse the existing system. Figure 21 presents the answers to this question.

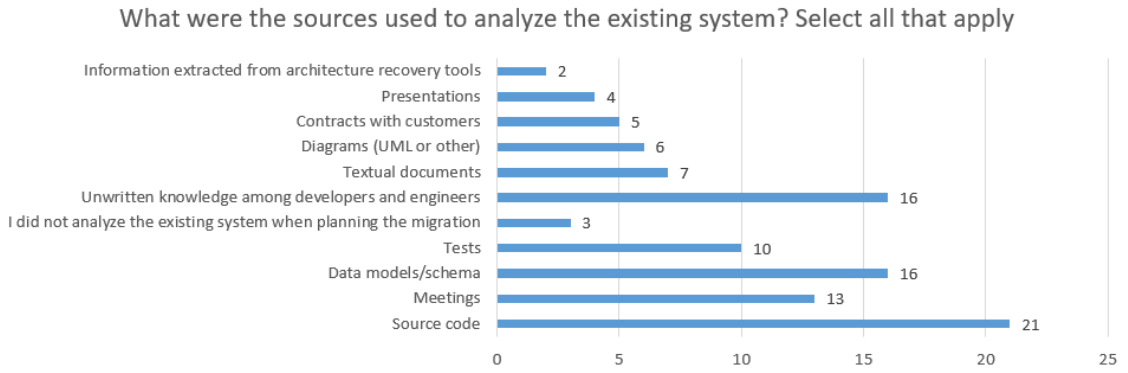


Figure 21 - Questionnaire- Sources used to analyse the existing system

Analysing this figure, it is possible to conclude that most of the participants were mainly focused in analysing the existing system through the source code as 21 out of 30 participants indicated that choice. Also, data models schema and unwritten knowledge among engineers was also a consistent choice with more than half of the participants indicating these sources. Meetings and tests also presented some relevancy with thirteen and ten answers, respectively. Finally, it is essential to notice that only 3 of the participants stated that they did not analyse the existing system. Therefore, it is relevant to understand the reason for this analysis, which is the focus of the next question.

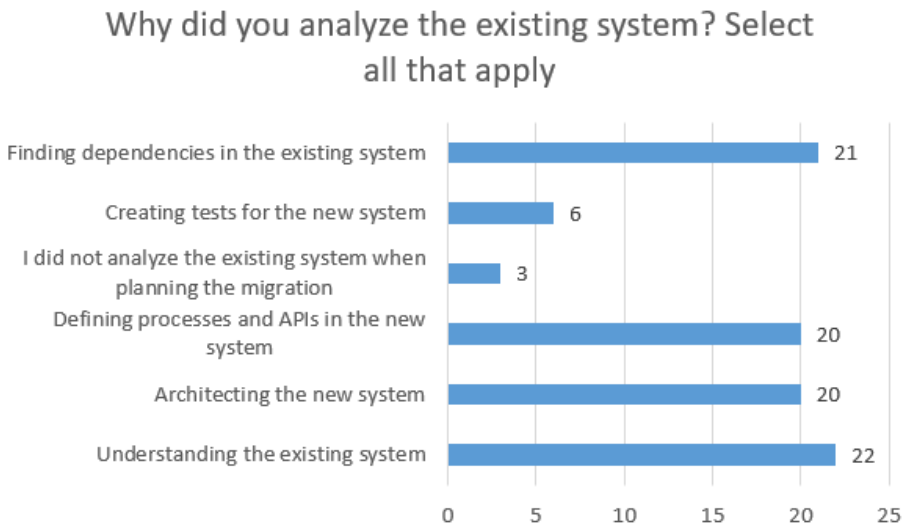


Figure 22 - Questionnaire - Reasons for analyzing the existing system

Figure 22 shows the reasons the participants stated for analysing the existing system. The answers remain consistent as the same three industry professionals indicated that they did not analyse the existing system. The answers of the remaining 27 participants are highly coherent as more than 20 say that the reasons were finding the dependencies in the existing system, understanding it, as well as defining processes and APIs in the new system and architecting it.

To finish this section, participants were asked what were the main challenges faced in this stage of the migration. The results are illustrated in Figure 23.

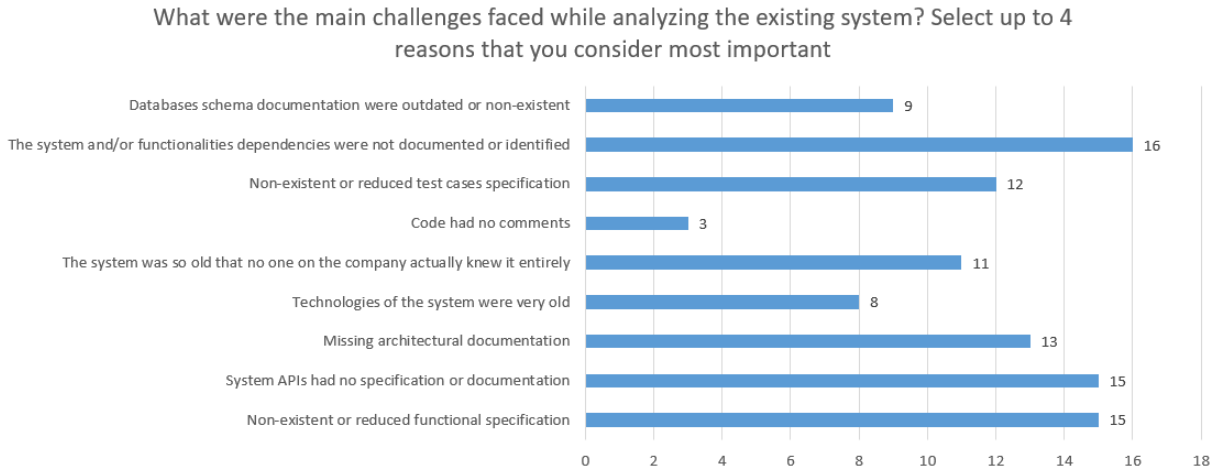


Figure 23 -Questionnaire - Main challenges faced while analyzing the existing system

The answers are distributed across multiple options, without a clear majority. However, the most selected options were all related with lack of documentation, either regarding functional characteristics, technical details or test cases specification.

6.3.3 Designing the new architecture

After analysing the existing system, the participants were asked some questions regarding the process of designing the new architecture. This section will analyse the answers to those questions.

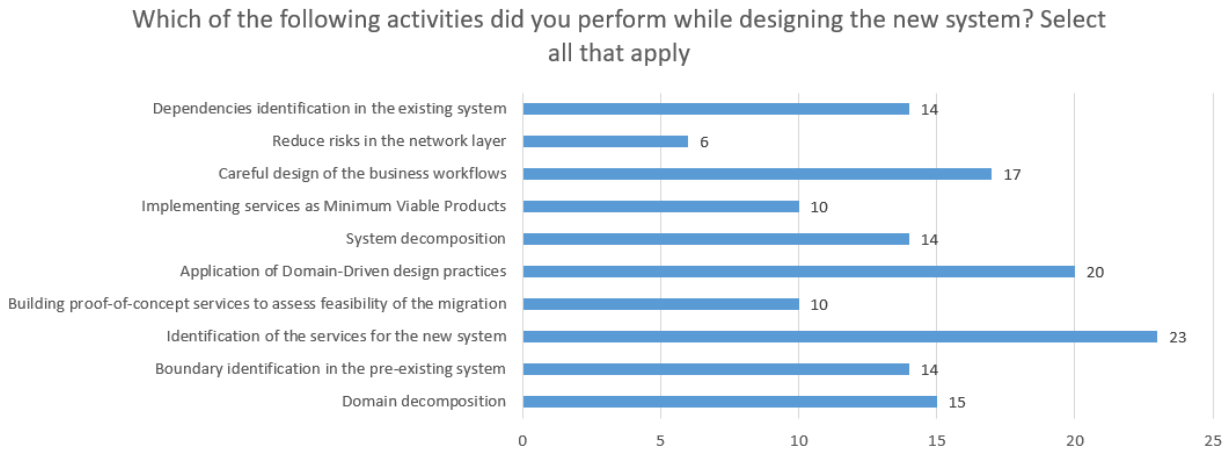


Figure 24 - Questionnaire - Activities performed while designing the new system

Figure 24 illustrates the activities performed by the participants while designing the new system. The answers go in line with one of the solutions identified in Section 6.2.5, as most participants selected choices regarding Domain-Driven Design, defining the proper granularity for the new services, and designing the business workflows accordingly.

As mentioned previously, to understand a system (either monolithic or microservices oriented), documentation is essential. Therefore, it is crucial that the new system is well documented to make it easier for future refactors to happen and even to expose the functionalities of the system in a clear way to clients or consumers of the services. For these reasons, the participants were asked in what way they documented the new architecture. The results are present in Figure 25.

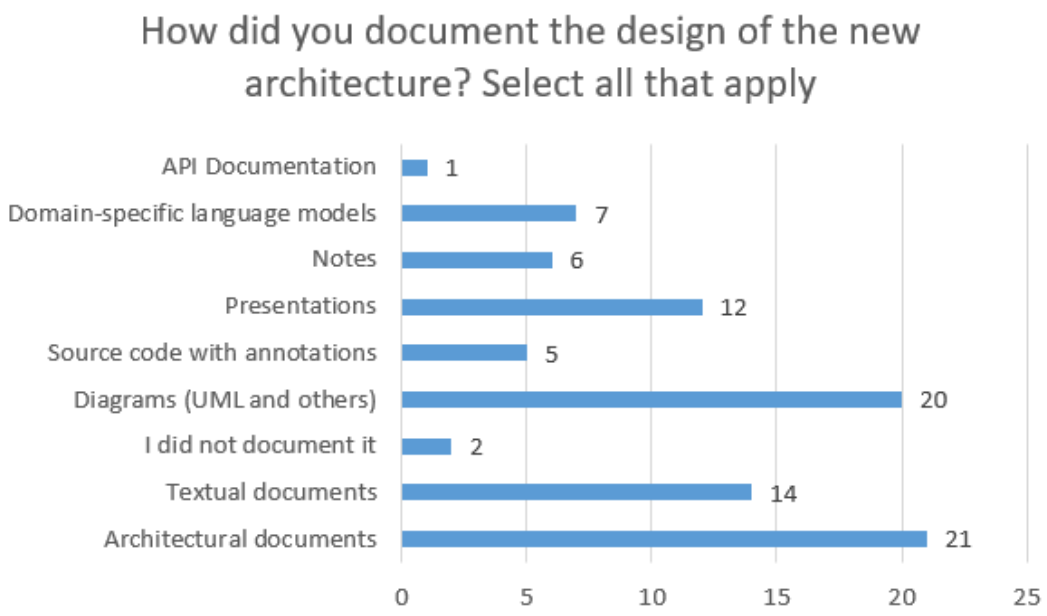


Figure 25 - Questionnaire - New architecture documentation method

In 30 participants, only 2 did not document the new architecture, which is a positive result. Most of the participants created architectural documents and diagrams to define the new architecture. However, only one of the participants mentioned API documentation, which may be an issue. It is also interesting to see that 7 participants mentioned Domain-specific language models to define the new architecture, which is a recent trend in software engineering that seems to be starting to be used in the microservices professional community.

The next question is not just about the design of the new system but also how the migration was planned regarding value delivery. Figure 26 shows the results of this question. 80% of the participants delivered new functionalities during the migration; only 16.7% of the answers state that new functionalities were not implemented. As mentioned in Section 6.3.1, most of the migrations reported in the questionnaire took more than half a year to complete. Therefore, it is justifiable that the migration also delivered new futures as most businesses are not able to completely stop for 6 months without direct value delivery, as the value that microservices provide can only be evaluated in the long term.

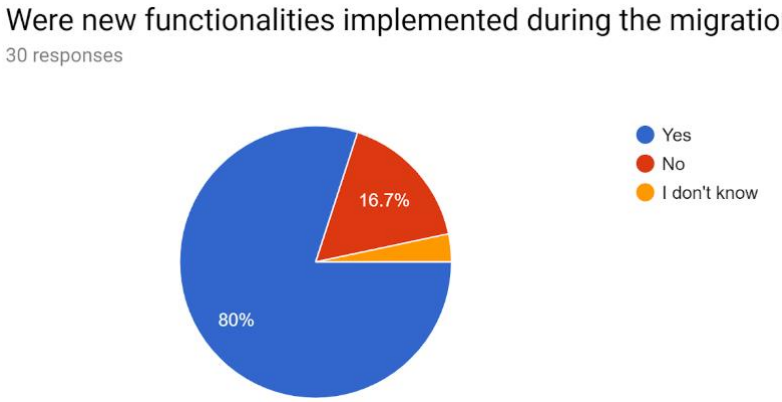


Figure 26 - Questionnaire - Value delivery plan for the migration

As a final question to this questionnaire section, the participants were asked what were the main challenges faced in the design stage of the new system. The results can be seen in Figure 27.

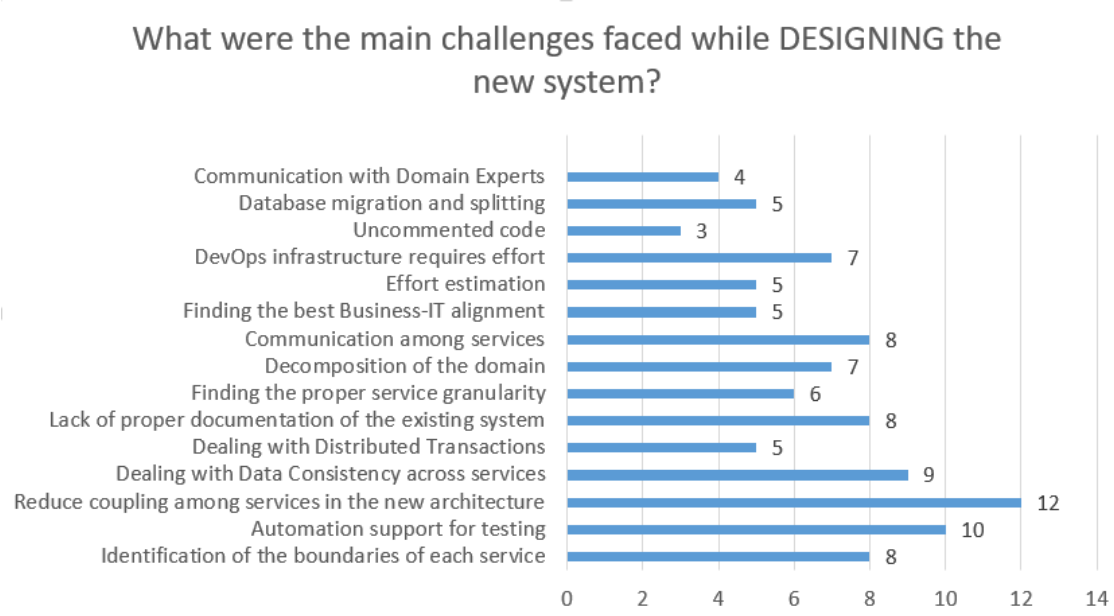


Figure 27 - Questionnaire - Main challenges of designing the new system

It is possible to notice that there are multiple answers referring to the decomposition of the existing system or of the business domain to model the new system components with the proper granularity. The most interesting conclusion of this question is that the three most selected answers are correspondent with three of the challenges identified in the Microservices Migration research, in Section 6.2.5, Table 19:

- Reduce coupling among services in the new architecture – is directly related with the challenged number 1 of Table 19: “Decomposition of the pre-existing system with the proper granularity and low coupling”.

- Automation support for testing – is directly related with “Testing Complexity on the Microservices architecture” identified previously, challenge number 4 of Table 19.
- Dealing with Data Consistency across services – is related with the distributed transactions challenge that this work addresses and was also identified in the literature study: “Data consistency issues”, challenge number 2 of Table 19.

6.3.4 Implementing the new system

This is the last section of the questionnaire referent to the migration process. It is focused on the implementation of the microservices architecture and migration execution.

The first question was about the strategy used to start the migration. As it can be seen in Figure 28, most of the participants started their migration by re-implementing existing functionalities as microservices. In Figure 26 we concluded that most participants delivered new functionalities during the migration, which means that even though the migration started by migrating the existing functionalities, during the complete process new features were also delivered in the microservices architecture.

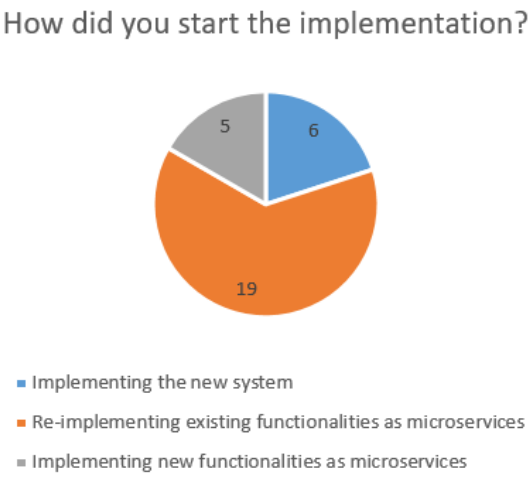


Figure 28 - Questionnaire - Migration strategy to begin the implementation

Either deciding to re-implement existing functionalities or implementing new ones, there must be a prioritisation of what should be migrated first. The second question of this section is precisely about that decision. The participant’s responses can be found in Figure 29. More than half of the participants started to migrate the functionalities with fewer dependencies, probably because those are the less coupled and therefore, it is easier to migrate them. However, the other participants used other criteria for this decision. Some stated that they started by the functionalities less used by the users, but there is also a participant who did the opposite and started with the functionalities that were most important for the users. Business flexibility and performance requirements were also mentioned.

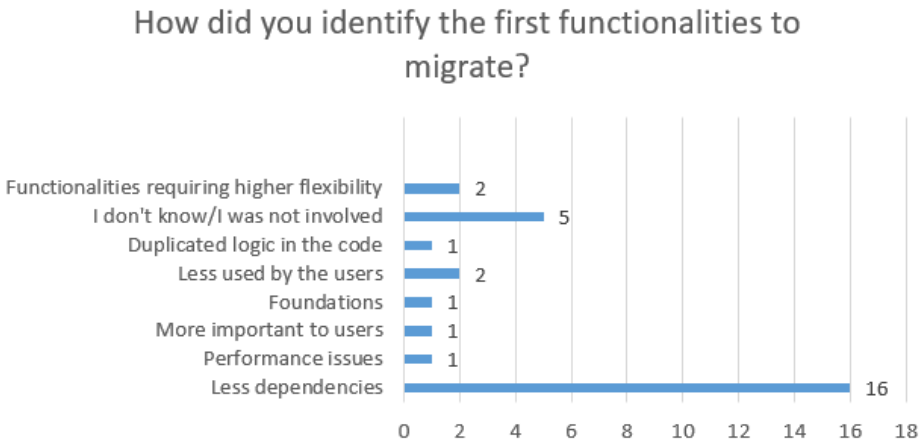


Figure 29 - Questionnaire - First functionalities to migrate strategy

After defining the criteria used to prioritise functionalities to migrate, the participants were asked what migration process was used to switch to the new system, among Parallel, Phased or Big Bang adoption:

- Parallel adoption: Having all functionalities writing information on both systems at the same time and reading operations only from one, being able to switch between the systems.
- Phased adoption: Having some functionalities on the new system and some in the old system.
- Big Bang adoption: Drop the old system and turn on the new one in a single step.

The results are shown in Figure 30. The most used process was phased adoption with 14 participants mentioning it, while 8 participants used the parallel strategy and only 3 used big bang adoption.

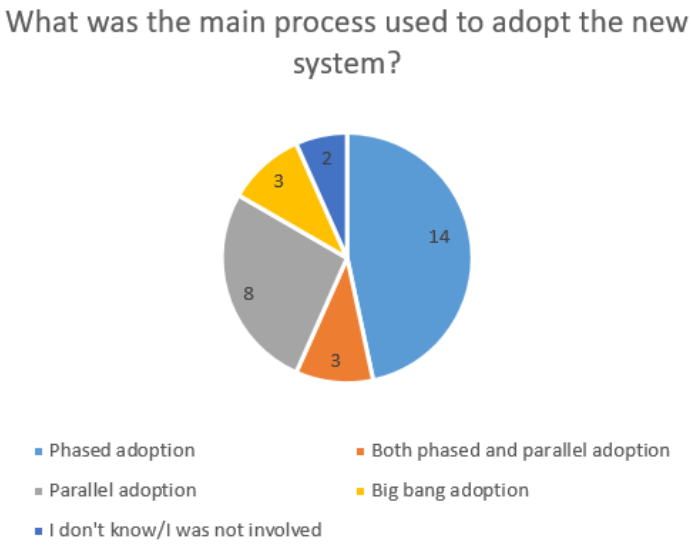


Figure 30 - Questionnaire - Migration process used to adopt the new system

It is also essential to notice that 3 participants mentioned that they started with phased adoption, but along the way, they needed to use parallel adoption for some features. No correlation was found between the adoption process used and the project delivery being delayed or not.

Then, the participants were asked how they handled data migrations in the process. The answers can be seen in Figure 31.

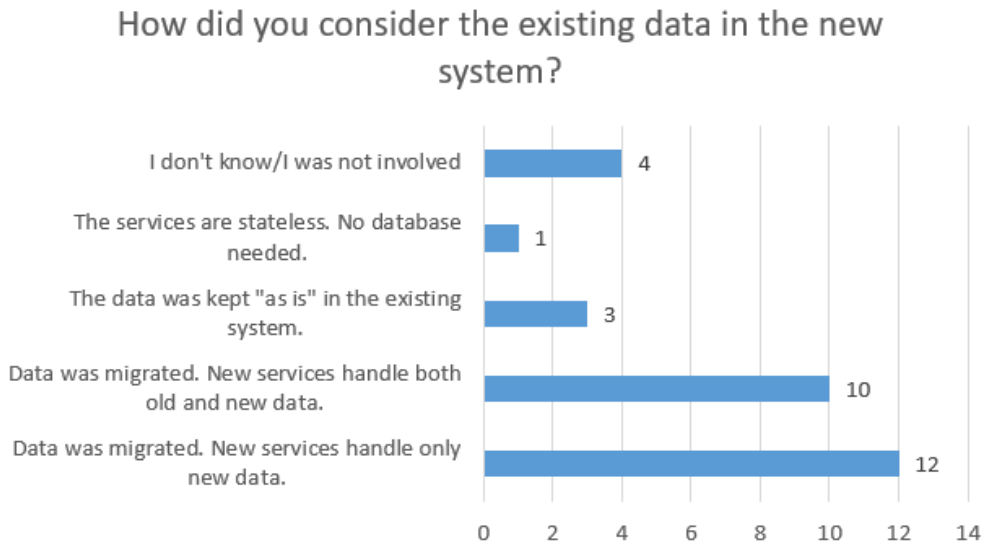


Figure 31 - Questionnaire - Data migration strategy

The majority of the participants (22) migrated their data to the new system. Twelve of them implemented the new system only capable of handling new data, while ten developed compatibility with old data schemas. No correlation was found between the data migration strategy and the adoption process mentioned in Figure 30.

The participants were then asked how many services they implemented in the new system. In the “Designing the new architecture” section of the questionnaire, participants were asked how many services they planned to have on the final system. The answers to both these question will now be analysed comparing the expectations in the design stage with the reality at the end of the migration. For this reason, the participants who are still in an early stage of the migration (less than 50% migrated) will not be considered as the final number of services may still change. The result of this analysis is present in Table 22.

Table 22 - Questionnaire - Planned Number of Services vs Final number of services

Planned number of services	The final number of services	Comparison
10	30	More services than planned
10	20	More services than planned
6	10	More services than planned
7	7	According to plan
150	350	More services than planned
50	50	According to plan
6	6	According to plan
12	14	More services than planned
10	10	According to plan
20	20	According to plan
15	15	According to plan
4	4	According to plan
30	35	More services than planned
20	20	According to plan
12	15	More services than planned
18	18	According to plan
10	10	According to plan
10	20	More services than planned
5	30	More services than planned
6	7	More services than planned
10	10	According to plan

With the data structured in Table 22 above, it is possible to visualise that some of the migrations reported had an increase in the number of services that were initially planned, and not a single migration had a decrease in the number of services. Figure 32 illustrates the percentages of each kind of comparison more clearly.

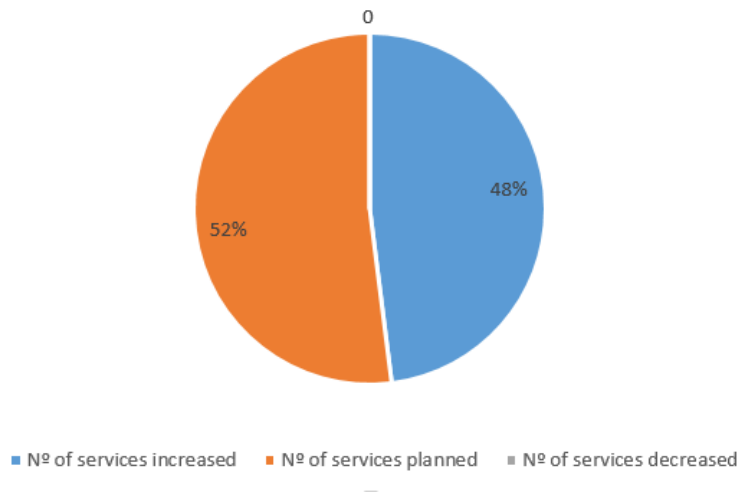


Figure 32 - Questionnaire - Planned number of services vs final number of services

To conclude the last section of the questionnaire regarding the microservices migration, the participants were asked what were the main challenges faced when implementing the system. The results can be seen in Figure 33.

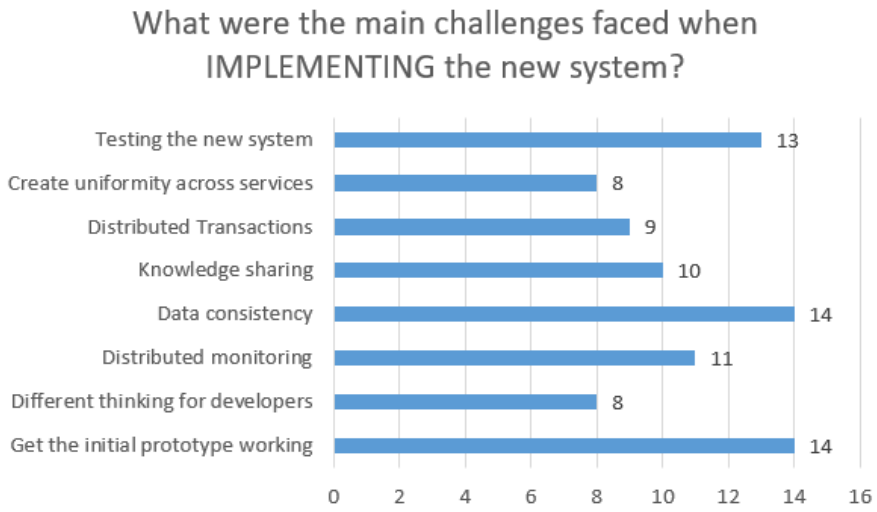


Figure 33 - Questionnaire - Main challenges faced when implementing the new system

Getting the initial prototype to work and dealing with data consistency were the two biggest challenges identified, which is consistent with the previous results of this work. “Testing the new system” was the second most selected challenge, which was also noticed in the systematic mapping study. Finally, distributed monitoring and transactions are the third most selected technical challenge. There were also some non-technical challenges identified that are worth mentioning, regarding knowledge sharing and handling different thinking (regarding the software architecture) for developers.

6.3.5 Questionnaire feedback

Finishing the questionnaire, the participants were asked to give some feedback regarding the questions and their experience participating in the study.

The suggestions received were regarding the size of the questionnaire. Some participants said that it should be more succinct, as some people may give up before completing it. Another feedback provided was regarding the different sections of the questionnaire: a participant stated that he got confused understanding the difference between the design and implementation stages sections.

6.4 Participant observation

The last component of the microservices migration research is a participant observation analysis of a real migration from a monolith to a microservices architecture done in a

professional context. This observation was done for several months, participating in software engineering activities in a team of engineers and architects, from design to deployment of the final solution.

As the project itself is confidential, only some technical details will be explained in order to bring value to the microservices field, and the solution will be described generically, as similar as possible to the real one.

6.4.1 Context

The initial system was a single monolith service that supported all the sections of the company business. The code base had really low maintainability, and any modification of the system often had unpredictable side-effects on other parts of the system. Furthermore, every time a new feature was added, regression tests had to be manually executed, which would take around a week to complete, and therefore the team was taking a long time to deliver new features. Also, a lot of the business logic was defined in database stored procedures, and with the increasing number of users, the system was not scalable enough to answer the business requirements. Furthermore, as the single monolith supported the entire business, there were multiple teams working in the service, which sometimes would create conflicts in deployments and version control, delaying value delivery.

For these reasons, the team decided to decouple the modules of its ownership to an isolated microservices architecture, which have more flexibility to business requirements and increased scalability and maintainability. As the company was aware of the mentioned issues, the management teams accepted the technological shift proposal, but as it would take around a year to complete the migration, some new features, bug fixes and increased performance through proper scalability had to be ensured. Therefore, with some small changes in the domain model, the team also delivered some features that were on the company roadmap.

6.4.2 Design of the new system

The team ownership supported a sub-unit of the business – named X from now on. There were some more experienced engineers in the team who had designed and implemented microservices in the past and that recommended to use Domain-Driven Design practices to define the new system boundaries and services granularity, oriented to the X Domain.

Following this suggestion, the team discussed how to design the new system, and some initial conclusions arise:

- Develop the new system following an incremental and iterative approach – Phased adoption;
- Define Boundary-Services: The system main purpose was to support X, and would, therefore, be a bounded context, following Domain-Driven Design concepts. Boundary-Services are responsible for managing communications with other bounded contexts.

- Define services for each sub-domain unit: Services who are the technical authority for a specific business capability inside a bounded-context.
- Use asynchronous messaging through a message broker to ensure communication between the multiple services: the reasoning for this decision was avoiding the temporal coupling that synchronous communication requires.
    - Using asynchronous messaging through a message broker, if service A needs service B and service B is down, service A can publish its message and continue to work as expected, when the service B comes back it will consume the message and resume execution.
- The architecture should be event-driven and reactive to event triggers.

With these principles in mind, the design of the new system was similar to the one illustrated in Figure 34.

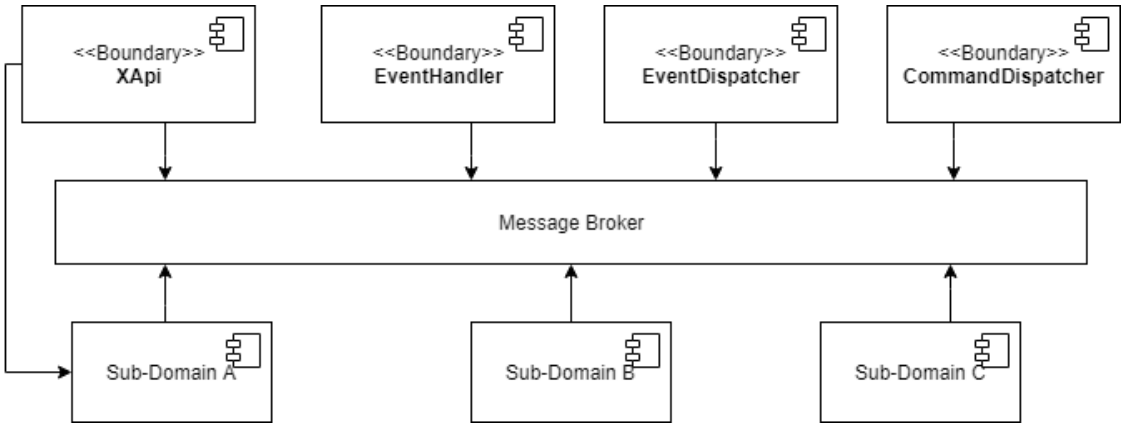


Figure 34 - Participant observation system high-level view

The following boundary services were then used:

- XApi provides a REST API for external communication. It is also an abstraction layer that converts REST requests into messages published to the message broker. In the case of a GET request, it accesses other APIs to obtain the information required.
- EventHandler is responsible for subscribing to external events from other bounded contexts. It works as an abstraction layer that adapts external events to an internal format and may add additional information required for the system execution. The internal message is published to the message broker, and any of the sub-domains can subscribe it and execute their own business logic.
- EventDispatcher is the third boundary service defined and exposes all the events triggered inside of the X domain so that other bounded contexts or external consumers can receive this information and react to it. It also works as an abstraction layer as it converts the format of the internal messages to an external contract.
- CommandDispatcher is the last of the boundary services, and its responsibility is to publish commands that will directly trigger operations provided by other bounded contexts.

6.4.3 Migration process

After defining the vision of the final microservices architecture and the principles to be followed, the team defined the migration process to be used. One of the principles defined was to migrate incrementally until the entire domain X was decoupled from the initial monolith (phased adoption). For this reason, the team decided to use the strangler pattern, however after some features had been migrated it was identified that some legacy systems were directly consuming data from the old system database which stopped being supported. Therefore, the team implemented the event decorating pattern to feed the new services data into the legacy database so that the other teams could have more time to adapt their systems to the changes and start consuming information from the services external API (XApi). For this reason, the migration process followed a phased adoption approach for some features and parallel adoption for others.

In the following sections, the strangler pattern and event decorating pattern usages will be described in more detail.

6.4.3.1 Strangler Pattern

Strangler pattern suggests the creation of an abstraction layer on top of both the monolith and the new system. This way, the consumers and clients are not affected for any changes below the abstraction layer, and the migration can be done incrementally with no impacts (Narumoto et al., 2017). An example of this approach can be found in Figure 35.

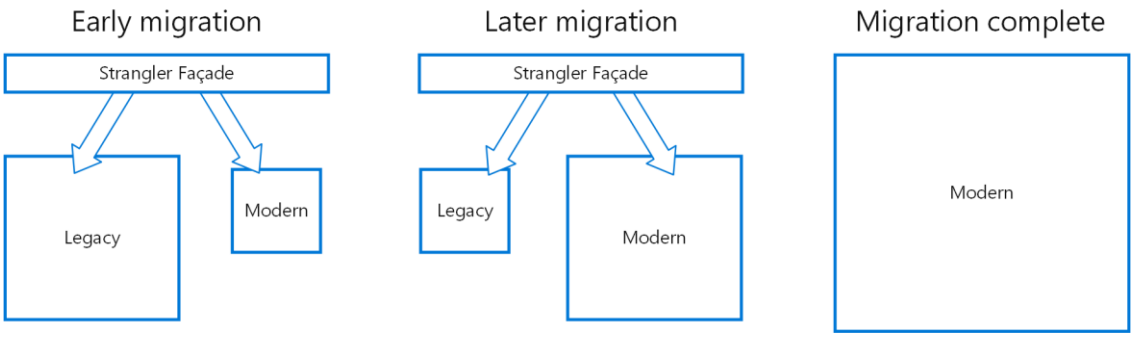


Figure 35 - Strangler pattern example (Narumoto et al., 2017)

Therefore, the first service of the new architecture was XApi, working as this abstraction layer and redirecting all the calls to the monolith initially. Defined the abstraction layer, the team started migrating an initial feature to the microservices architecture. At first, XApi was redirecting write operations to both the legacy and the new system; however, read operations were only performed in the legacy system. This allowed the team to keep the system behaviour unchanged, while still feeding data to the new system in the production environment. With this approach, the team was able to observe the new system and ensure that it was providing the same behaviour as the monolith, and the information was consistent between the two. Also, the new service could be tested under production environment heavy load before being truly used by the final user. When the team was confident regarding all the requirements of the new system, the read operations were redirected to the new service, and the monolith was no

longer supported. The same approach was followed for the other features, components or entities of the monolith until the migration of domain X was complete.

6.4.3.2 Event Decorating

As previously mentioned, after some features were fully migrated and data was no longer inserted in the legacy database, it was identified that some other monolith components were tightly coupled to those database tables. Therefore, temporarily, the legacy database had to be supported so that other teams would not be impacted. To achieve this, the team implemented the event decorating pattern, as illustrated in Figure 36.

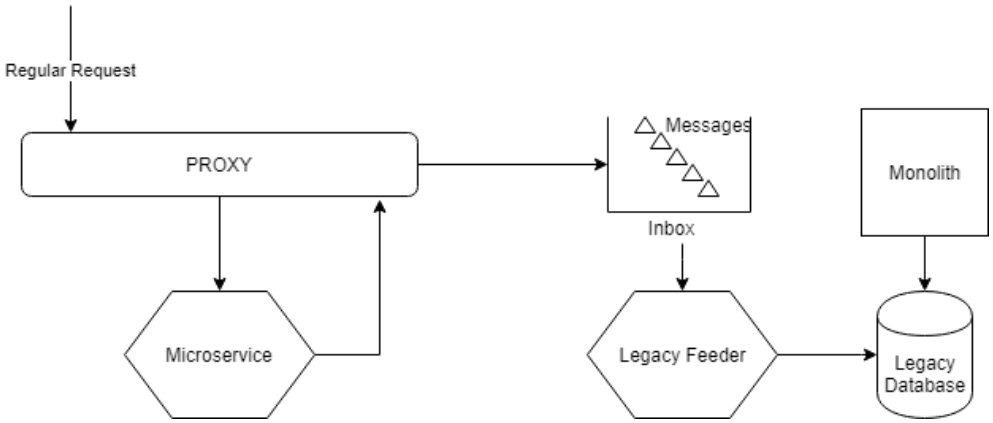


Figure 36 - Event decorating example

The new system services are identified by a hexagonal shape, while the old monolith is a square. The responses of the microservices were subscribed by a proxy layer that published messages for each one of the responses to a message queue “Inbox”. This message queue was consumed by the “Legacy Feeder”, which was implemented with the sole responsibility of listening to these messages, writing the changes into the legacy database and executing stored procedures when required ensuring that the data was coherent with the legacy behaviour. With these solutions, the other teams were able to rely on the legacy database until they adapted their systems to the new microservices architecture of domain X.

6.4.4 Monitoring

The initial monolith had a single dashboard with all its logs, and it was possible to understand what was happening in the system. However, the new system has more than 30 services and having a different dashboard for each one would not be viable. Furthermore, it would not be possible to have a global vision of what was happening in the system. For these reasons, the team used the log aggregation concept and built a dashboard containing the log information of all the services combined. To be able to manage all this information, all the system messages are logged and contain mandatory properties:

- A correlation identifier: Single identifier of a business process instance. All the messages flowing in the system to achieve a specific instance of a business process have the same

correlation identifier. It is generated by the service that initiates the business process and is redirected by all the participant services. This allows the team to visualise the flow of messages through all the services, and to identify where a possible issue exists if the business process fails.

- A process name: A keyword that identifies the functionality that a specific message is trying to achieve. It is not necessarily the name of the business process, as it regards a specific microservice functionality. Each microservice must log the beginning and the end of the process. When the end is not successful, the microservice will log the end of the process with a warning or as an error. All these logs contain a correlation identifier.
- EntityId: Identifies the domain entity affected by the message, so that it is possible to track changes to a specific entity instance to troubleshoot possible issues.
- Timestamp: Date and time when the log happened.

6.4.5 Testing

Testing was one of the hardest challenges the team faced during the migration. Only a few unit tests were implemented in the first version of the microservices oriented system. Testing was not taking into consideration when planning the migration and it was forgotten. The team quickly understood this was a mistake as the delivery of new features was taking a long time as manual regression tests still had to be performed. Furthermore, these manual tests in the microservices architecture are much more complex and take more time. Because of this mistake, one of the main benefits of microservices – fast delivery - could not be achieved.

To solve this problem, the team started implementing automated tests to cover existent features. First, unit tests with a code coverage percentage of more than 80% were implemented. Then component tests – black-box tests for a specific microservice - were developed, allowing the quality assurance engineers to specify test cases through JSON notation. In the future, the team plans to develop automated integration tests between the services that constitute sub-domains of X, and use the concept of consumer-driven contracts to develop tests between sub-domains.

6.5 Results summary

In the previous sections of this chapter, both literature and industry data were retrieved and analysed, concluding the obtained results. This section has the objective of compiling all the information of this chapter sections in a summary result which can be presented to microservices researchers or practitioners in order to provide a catalogue of common challenges currently faced, and some solutions that are currently being used for those issues.

6.5.1 Technical challenges and solutions catalogue

This section describes the technical challenges and correspondent solutions identified in the microservices migration research. All the issues will be defined by presenting the problem context, description, solution and origin.

6.5.1.1 Data consistency and distributed transactions

Problem context: Working with a monolith, usually teams have a single database where all the transactions are applied or roll-backed if there is an error in the middle. When teams move to the microservices architecture, they can no longer ensure the same ACID transactions as the original data schema is decomposed in multiple services, most of the times each service with its database.

Problem description: Ensuring consistency across the multiple databases that now exist in the microservices system, and managing the distributed transactions that are now executed across multiple services in order to fulfil a business process.

Solution: The two-phase commit should be avoided to benefit from the microservices architecture advantages, as it would impact the performance and availability of the system. Microservices were not designed for strict consistency requirements, and are therefore only advised when eventual consistency is an option. Therefore, to handle data consistency and distributed transactions in a microservices system with eventual consistency, a standard solution is the implementation of the saga pattern to manage the distributed transactions and ensure data consistency by executing compensation actions when there is a failure in the middle of the business process.

Origin: Systematic mapping study, Industry questionnaire and participant observation.

6.5.1.2 Testing Complexity

Problem context: In a monolithic system, teams usually implement unit tests and automate tests validating the integration of the application with database systems and other infrastructural dependencies in order to verify if the business logic implementation is working as expected. In a microservices system, there are now multiple services to test, and the communication between them also needs to be validated.

Problem description: While in a monolith system, a single service needs to be tested, in the microservices architecture, there are multiple services to be validated. Furthermore, the integration among them also needs to be verified, which naturally increases the testing complexity of the system.

Solution*:* A way to handle the increased complexity of the tests is to automate them in a continuous integration pipeline, following DevOps principles. Also, integration tests between different services should apply to specific bounded contexts, following Domain-Driven Design concepts. Consumer-driven contracts testing may be used to validate the communication

between these different boundaries, in order to ensure that the communication contract between the two parts remains valid.

Origin: Systematic mapping study, Industry questionnaire and participant observation.

6.5.1.3 Setup and execution of the initial prototype

Problem context: The initial setup of the microservices architecture demands more effort than a monolithic system. The microservice architecture implies that multiple services are developed, deployed and executed in the production environment. With a monolith, this is simpler as there is a single service to setup and execute.

Problem description: As there are more components in a microservices architecture than in a monolithic one, the initial setup of the system may be more complex. This phenomenon was mentioned by authors and named “MicroservicePremium” (Fowler, 2015b).

Solution: When the microservices architecture is being developed from scratch the team should try to use the monolith-first approach instead. Monolith-first suggests that even if an application use case seems appropriate for the microservices architecture, the team should first implement a monolith instead, and migrate it to the microservices iteratively as its complexity and component boundaries become well defined, as Microservices are mostly useful on more complex systems with well-defined boundaries (Fowler, 2015a).

Origin: Industry Questionnaire.

6.5.1.4 Creating uniformity across multiple services

Problem context: Microservice architecture implies that multiple services work together to achieve a common objective. For this reason, teams must know the implementation details of all these systems in order to define and implement new features.

Problem description: When there is no coherency, uniformity or standardisation across the multiple services, their details become different. Therefore, it becomes harder for a software engineer to move from one service to another while implementing new features.

Solution: A solution for this issue is to define coding and implementation conventions and specify them through Model-Driven Software Engineering or defining a Domain Specific Language, which allows the team to implement code generators that ensure uniformity across multiple services. Therefore, the services would be more coherent. Also, static code analysis tools can be used to ensure the fulfilment of the defined specification by all the implemented services.

Origin: Industry Questionnaire.

6.5.1.5 Distributed monitoring

Problem context: In the microservices architecture, the business processes may require multiple services to be fulfilled entirely. For this reason, to monitor the business process success, the entire distributed system must be monitored.

Problem description: While in a monolith there is a single source of information to monitor, in a distributed system there are multiple systems providing valuable information. Teams must implement a way to easily visualise the data in order to be able to understand what is happening in the system and be alerted if an error happens.

Solution: To mitigate this difficulty, the concept of Log aggregation should be used. It consists of aggregating all the logs of multiple services in a single dashboard in order to centralise the visualisation of what is happening in the system. These logs should contain valuable information such as the process name, information specific to the business process, to the entity affected and correlation data. Furthermore, all the messages should have a correlation identifier. A correlation identifier is a unique key passed through all the messages required to fulfil a business process instance. In this way, it is possible to visualise the messages flowing in the system and identify where a possible error happened using monitoring and tracing tools.

Origin: Systematic mapping study, Industry Questionnaire, Participant Observation.

6.5.1.6 Decomposition of the pre-existing system with the proper granularity and low coupling

Problem context: The existent monolith supports an entire business model in a single executable component, usually with a single database schema for persistence. When adopting the microservices architecture, one of the first steps is to decompose this single piece into multiple services, each with its well-defined boundaries, in order to achieve low coupling in a distributed system modelled around a business domain.

Problem description: When the responsibilities and boundaries of the new services are not well defined, the services are not independent. For this reason, microservices are unable to provide the business flexibility that the architectural style proposes. When this happens, usually teams end the migration with a “distributed monolith” – a distributed system in which the different components are highly coupled and dependent on each other.

Solution: Most successful microservices migrations identified used Domain Driven Design to deal with this issue. Microservices should be independently deployable but work together to achieve a common goal, modelled around a business domain. Domain Driven Design concepts define the design of the system aligned with the domain model, specifying bounded contexts and entities that are easily mapped to physical components of the microservices architecture. Furthermore, the use cases and business workflows are defined across the multiple identified entities or domain aggregates, and therefore the technical authorities for those specific business capabilities are mapped in specific physical components with the proper granularity.

Origin: Systematic mapping study, Industry Questionnaire, Participant Observation.

6.5.2 Migration approaches catalogue

As a result of the research, some migration patterns and approaches were also identified, but do not fit the catalogue of technical challenges presented in Section 6.5.1. However, they may be useful in the planning of the migration technical execution in order to do it in an efficient way with reduced errors possibility.

Three migration approaches were identified:

- Big Bang adoption: The results of this work research conclude that this is the less used approach, but it was mentioned from some sources and should, therefore, be considered. Using Big Bang adoption, the team shuts down the existing system and enables the new microservices architecture in a single step. This approach was considered to be the most dangerous one as if the architecture is not stable and has any mistake, the entire system will collapse and there may be data corruption caused by the new system errors. Therefore, when using this approach, there should be strong and solid confidence in the new system, and a contingency and rollback plan should be defined.
- Parallel adoption: Parallel adoption suggests that functionalities are implemented and enabled in the new system, without shutting down the old system. This is achieved by having both systems answering write operations, while the read operations are easily redirected between the two systems. The essential difference between parallel and big bang approach is that parallel adoption allows the team to validate the consistency of the new system by comparing the existing system behaviour – which is the expected one as it was the one previously provided - with the microservices architecture output. This can be visualised in the microservices responses or data. During this validation stage, the read operations are answered from the monolith as expected. When the team is confident regarding the new system, then read operations can be redirected to it.
- Phased adoption: This migration approach focuses on moving some functionalities to the new system while other functionalities remain in the existing system. This approach increments value to the microservices architecture iteratively. It allows the team to evaluate the quality of the solution faster while improving the development process along the way. The repetition of mistakes is avoided when using this approach. Furthermore, most of the errors are centralised and isolated in smaller parts of the overall system. For this reason, the first parts migrated are more prone to errors. Therefore, a prioritization of what the first functionalities should be is essential. In this work research, it was identified that there are multiple criteria used by teams for this prioritisation: functionality relevancy to the stakeholders, components of the system with fewer dependencies, number of users of specific functionality, among other criteria that were less mentioned by research participants and the analysed literature.

Phased adoption can be combined with the other two mentioned approaches. These combinations will now be better explained.

- Phased + Parallel: When combined with parallel adoption, each one of the sub-parts or functionalities defined by the phased adoption prioritisation are migrated in parallel as described above in “Parallel adoption”.
- Phased + Big Bang: Combined with Big bang adoption, each one of the modules migrated in each iteration of the phased adoption is switched in a single step without a previous parallel validation.

6.6 Threats to validity

Naturally, there are some threats to the validity of this work, which are described in this section.

First of all, some important information might be missing in the reports from the industry survey. The cause for this is the static nature of the questionnaire (the same for all participants), which limits the acquisition of information specific to each participant. Also, some participants might have misinterpreted specific questions and therefore providing invalid answers.

Another threat to validity is the number of answers to the industry survey. As mentioned in 9.2, if the survey did not have so many questions there would probably have been more participants. If more answers were provided, a more geographically distributed study would be possible, which would provide more valuable insights regarding the microservices topic.

Regarding participant observation, there is always the possibility of bias as the researcher actively participated in the migration and may have been influenced by his perception of the project.

Furthermore, the systematic mapping study is also influenced by some kind of bias as the classification system used (Section 6.2.3), inclusion and exclusion criteria (Section 6.1.3.1), and overall research plan (Section 6.1.3) is naturally influenced by the researcher experiences.

7 Distributed transactions solution

After concluding the research in microservices adoption most common challenges and best practices, this chapter aims to present a solution implemented to solve one of the identified challenges: Distributed transactions management.

7.1 Analysis

One of the goals of this work is to support the software engineering community with a solution for the distributed transactions challenge of microservices migrations. Therefore, this section will analyse possible design alternatives and requirements for the final solution.

7.1.1 Context

Section 4.2 analysed different existent approaches for implementing distributed transactions. From that analysis, the conclusion is that saga pattern is the chosen approach to apply in this work. The reason for this is that even though both strategies support failure-handling mechanisms, 2PC compromises the system availability and performance while sagas handle failures using compensating actions without locking the system resources, based on eventual consistency. One of the main benefits of microservices is availability. Therefore, sagas are the chosen approach in this context, which is the focus of this work. There are two ways to implement the saga pattern, using orchestration or choreography:

- When using orchestration, having a separate service makes it easier to provide visibility over the distributed transactions happening in the system, and the effort of adapting the solution to different implementation scenarios is reduced as most of the logic is implemented in a single new component. Also, using orchestration, it is easier to control synchronous operations of request/reply and change the order in which they must be executed (Bonham, 2019). However, the orchestrator service becomes highly coupled to all the services it orchestrates, becoming a single point of failure for the

business processes it operates. Furthermore, this approach introduces the risk of isolating too much logic in a single service over time, creating anaemic CRUD-based services that are managed by a monolithic orchestrator service (Newman, 2015). When that happens, the system is going against one of the microservices principles *"smart endpoints and dumb pipes"* (Fowler and Lewis, 2014).

- The choreography approach is significantly more decoupled as any service can publish events to an event stream and be plugged to the event stream to subscribe to events from any other service. When there is the need to implement a new process, a service only needs to be plugged into the event stream to consume the required events. Therefore, following this approach, the services are decoupled from each other, without a single service coordinating every step of the process. The tradeoff is that as the transaction logic becomes distributed across the multiple services participating in the process, the view of the business process becomes only implicit across the system without a single point to be accessed explicitly. However, to mitigate this issue, monitoring practices can be implemented, allowing professionals to visualise all the events that are being published and consumed across the microservices architecture and creating the explicit image of the business process (Newman, 2015).

The described solutions are viable for different use cases, and both are used in the industry. Therefore, teams implementing the saga pattern should evaluate their use case to understand where they should use orchestration or choreography. However, as described in Section 4.3.2, there is no implemented solution to help teams implement choreographed sagas. All the identified solutions are highly focused on orchestration and workflow rules management. For this reason, this work contributes to the field by providing an implementation of choreographed sagas that facilitates the usage of this approach when teams decide that it is the right approach for their use case.

The developed solution has the primary purpose of helping software developers to implement sagas in their microservices architecture. As the library follows the choreographic approach, it was entitled *Sapher* – a mix of the words Saga and Choreographer.

7.1.2 Domain model

A business process consists of multiple steps. In a microservices architecture, each one of these steps is usually executed in different services. Therefore, a business process step communicates with others using messages. Services implement message handlers for the messages that they need to act on to accomplish the mentioned interaction between steps.

These messages can be considered input messages when they start a business process step or response messages when they are a response to a previously made request or a notification that a step needs to receive to ensure the success of the distributed transaction.

Sapher domain model can be visualised in Figure 37 below.

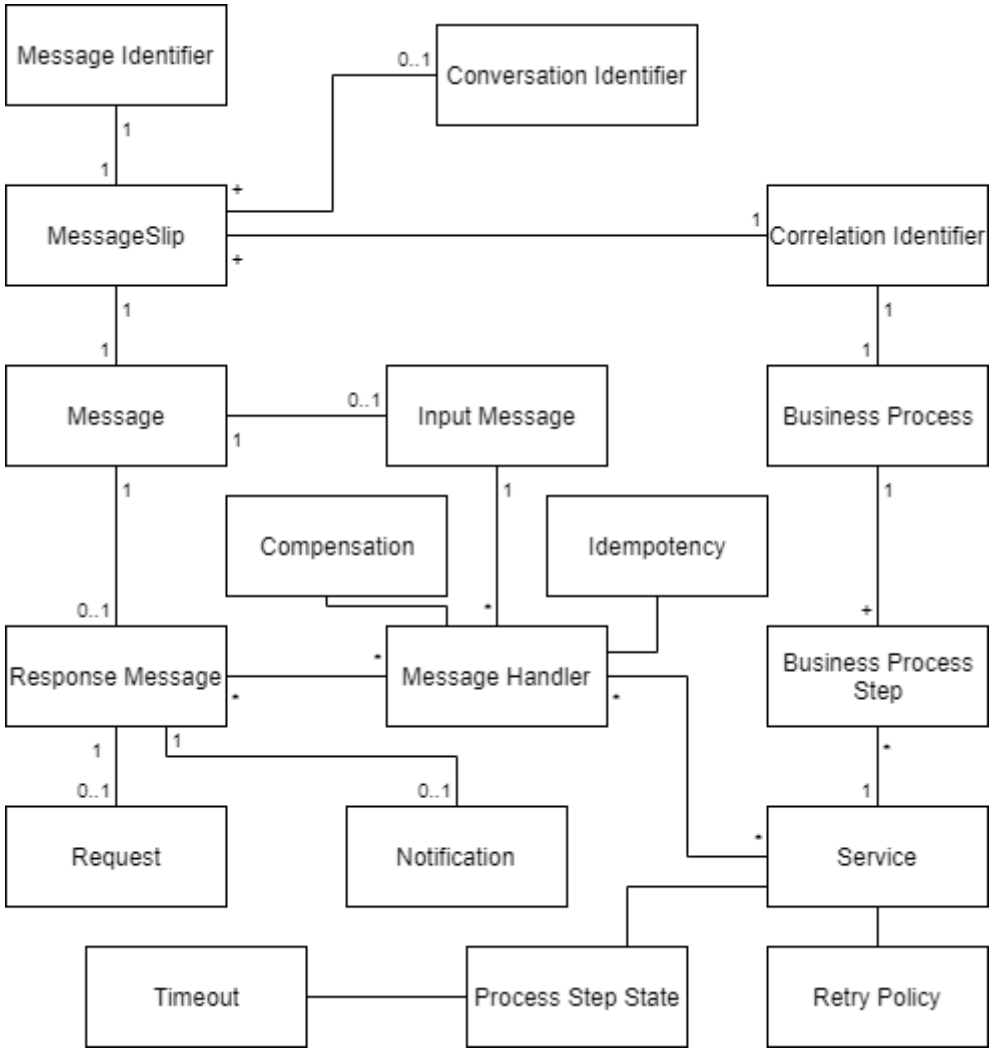


Figure 37 - Sapher domain model

Also, the message handlers are idempotent so that message producers can use retry policies to ensure the correct delivery of messages, supporting a more resilient system with more reliability in business processes. For that reason, all the messages are associated with a MessageSlip that contains the following information:

- Message identifier – a unique identifier of a message;
- Correlation identifier – a unique identifier of the transaction, and therefore of the business process instance;
- Conversation identifier – the message identifier of the previous message in the transaction.

Furthermore, message handlers also provide compensation for messages that require it. Each process step state can be considered as failed after a specific time interval without being updated, which is considered a timeout.

Also, each process step can have multiple handlers, and each handler can be assigned to multiple steps in order to achieve a high reusability of business logic. Finally, a message can have multiple handlers, depending on the process step they are being applied to.

7.1.3 Requirements

This section describes the identified requirements for this solution and is divided in non-functional and functional requirements.

7.1.3.1 Non-functional requirements

As this work focus is on the microservice architecture style, the provided solution must respect its patterns and guidelines. As concluded in Section 7.1.1, this work intends to contribute to the field by providing an implementation of choreographed sagas, which is another requirement for the final solution.

Furthermore, the technologies described in Section 4.3.2 showed most value when the implementation did not require structural changes to the user code and was agnostic to communication channels. Finally, failure-handling mechanisms should be supported. Table 23 presents the identified non-functional requirements for this solution.

Table 23 Distributed transactions solution non-functional requirements

Requirement number	Description
1	Provide reduced effort in adapting the solution to different implementation scenarios.
2	The microservices architecture patterns and guidelines must be respected.
3	Usage of choreographed sagas
4	The solution must be agnostic to communication channels.
5	The solution must provide failure-handling mechanisms.

7.1.3.2 Functional requirements

Sapher was created to help developers, and therefore, all the use cases have the same actor: A developer trying to implement distributed transaction in a microservices architecture.

In Figure 38, the identified use cases are illustrated in a use case diagram.

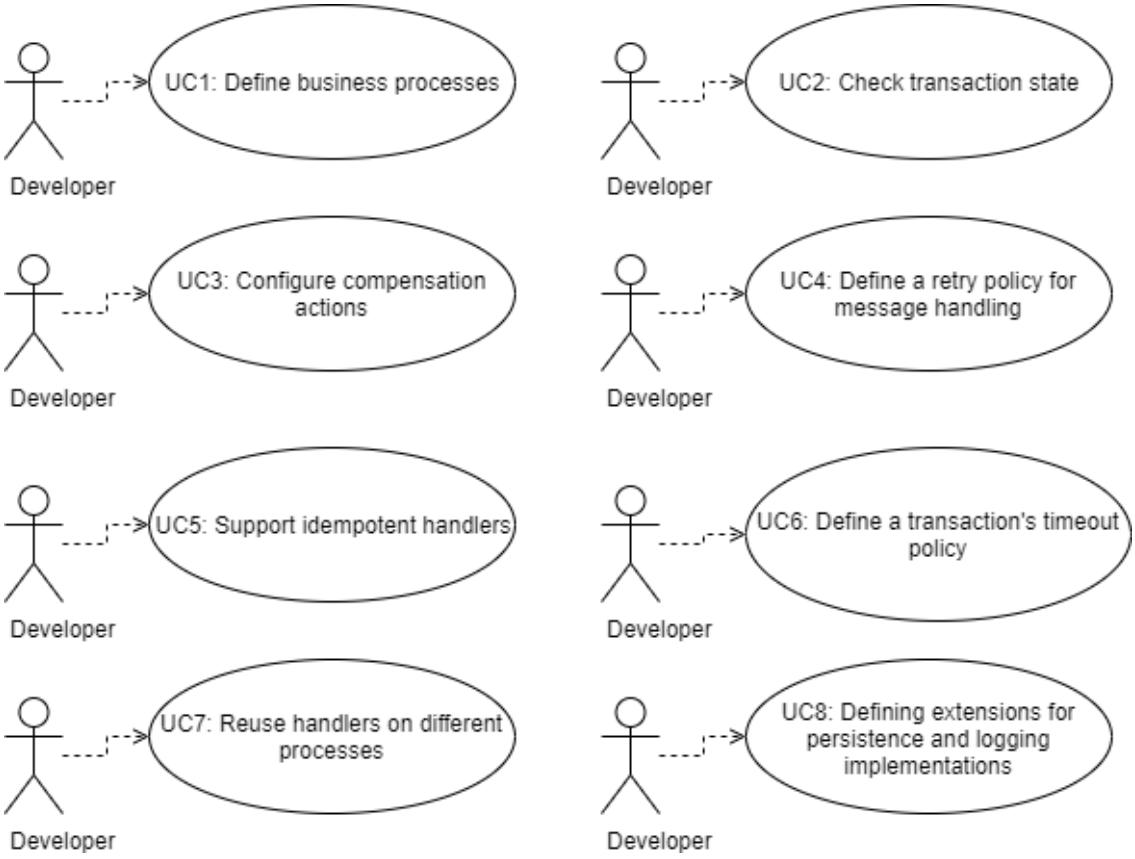


Figure 38 – Sapher use case diagram

As previously explained in this document (see Section 4.2), a distributed transaction is a business process spanning across multiple services. Therefore, its definition is essential for the implementation of distributed transactions, resulting in UC1.

Also, it is essential to control the state of the transaction so that actions may occur when anything fails, and the user can understand what is happening in the system. UC2 provides this functionality.

UC3 represents the compensation functionality that a saga implementation must provide.

To ensure no messages are lost, retries are essential in a distributed system, which is the reasoning for UC4. However, to be able to do that, the consumers must be idempotent, resulting in UC5.

In a distributed system, when a requesting service expects asynchronous replies, but the response never reaches the requester, the transaction is as failed. UC6 provides this possibility by allowing the user to define a time interval to wait for the response before marking the transaction as failed.

UC7 and UC8 are related to maintainability, reusability and flexibility of Sapher. To avoid duplicated code across different processes, the user can reuse the same message handlers in

different business processes. Also, the user can implement their logger and persistence logic as long as they follow the extensibility contract defined by Sapher.

7.1.4 Design alternative

Analysed the possible strategies and the defined requirements, it was determined that the choreography approach should be used as there is no solution to aid in the implementation of choreographed sagas. The alternative would be to use orchestrated sagas. Therefore, this section contains a high-level description of what that alternative would be.

The high-level view of the final solution is defined in Figure 39.

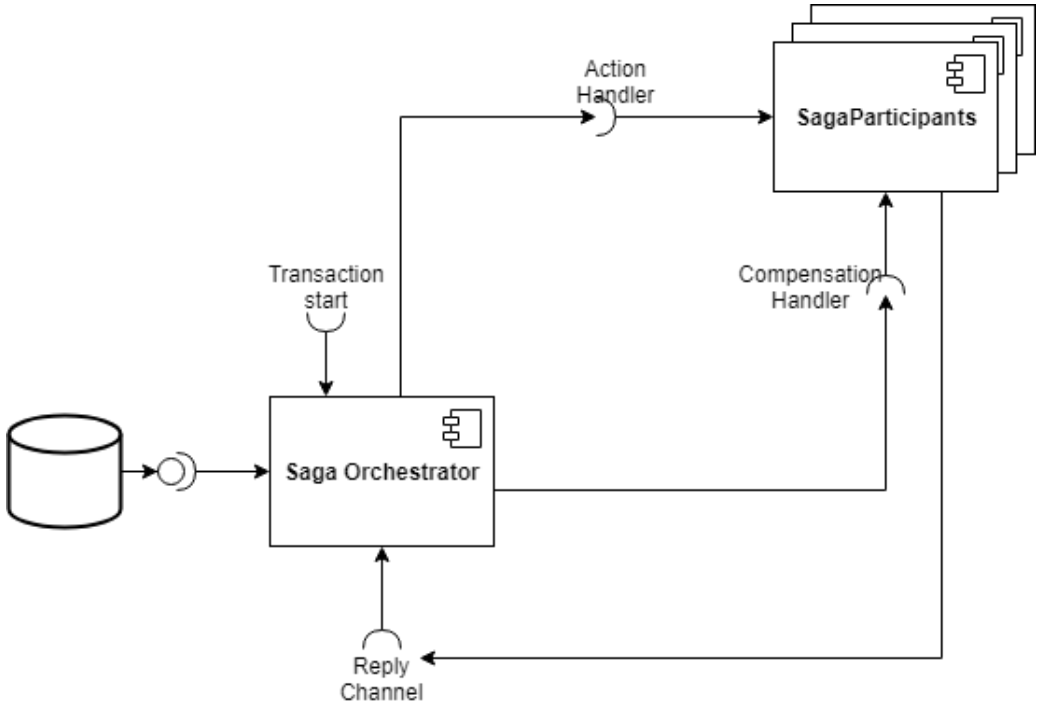


Figure 39 – Orchestrated sagas solution high-level view

The solution provides implementations for the two types of actors of an orchestrated saga process, the orchestrator and the participant. The orchestrator offers an endpoint to start the transaction and another to receive replies from the participants, which provide endpoints to execute regular actions or compensation actions. Furthermore, the orchestrator records the saga transaction state in a database with the data model described in Table 24. This information is accessed via a web application provided by the saga orchestrator, providing visibility over the system distributed transactions.

Table 24 – Data model for saga transaction

Property name	Property type
TransactionId	UUID
TransactionName	String
Status	String
Reason	String
Step	String
UpdatedDate	DateTime

TransactionId is a unique identifier generated by the orchestrator for each transaction. TransactionName is a label provided for the transaction. Status is a pre-defined range of Strings (Started, Aborted, Complete, and Failed). Reason only has value when the Status is failed and indicates the cause of failure. Step describes the label of the step in which the transaction is, and UpdatedDate provides information regarding the last time the object was updated. This model should be persisted in a non-relational database as it consists of a single entity with no relationships.

The solution provides two distinct libraries to be used by the saga orchestrator service and the saga participants respectively. The saga orchestrator library provides an abstract class SagaOrchestrator which can be inherited to define endpoints for the transaction start and reply channel. Furthermore, the implementation of SagaOrchestrator must define its Workflow using the WorkflowBuilder class, which provides functionalities for identifying the different steps of the transaction and compensation action for each one of them. It also defines the expected replies for each one of the outlined steps and which ones should be considered successful. The library is responsible for abstracting the coordination between the different steps defined, execute compensation actions, and record the transaction state.

The SAGA participant library provides a SagaParticipant abstract class which can be implemented to define endpoints for action and compensation handling. After the execution of the action successfully, the SagaParticipant automatically informs saga Orchestrator of the success.

7.2 Design and implementation

This section describes the solution and provides implementation details for the major components and use cases it supports.

7.2.1 Logical view

Defined the domain model and the use cases to be addressed, Sapher was designed following the high-level design illustrated in Figure 40.

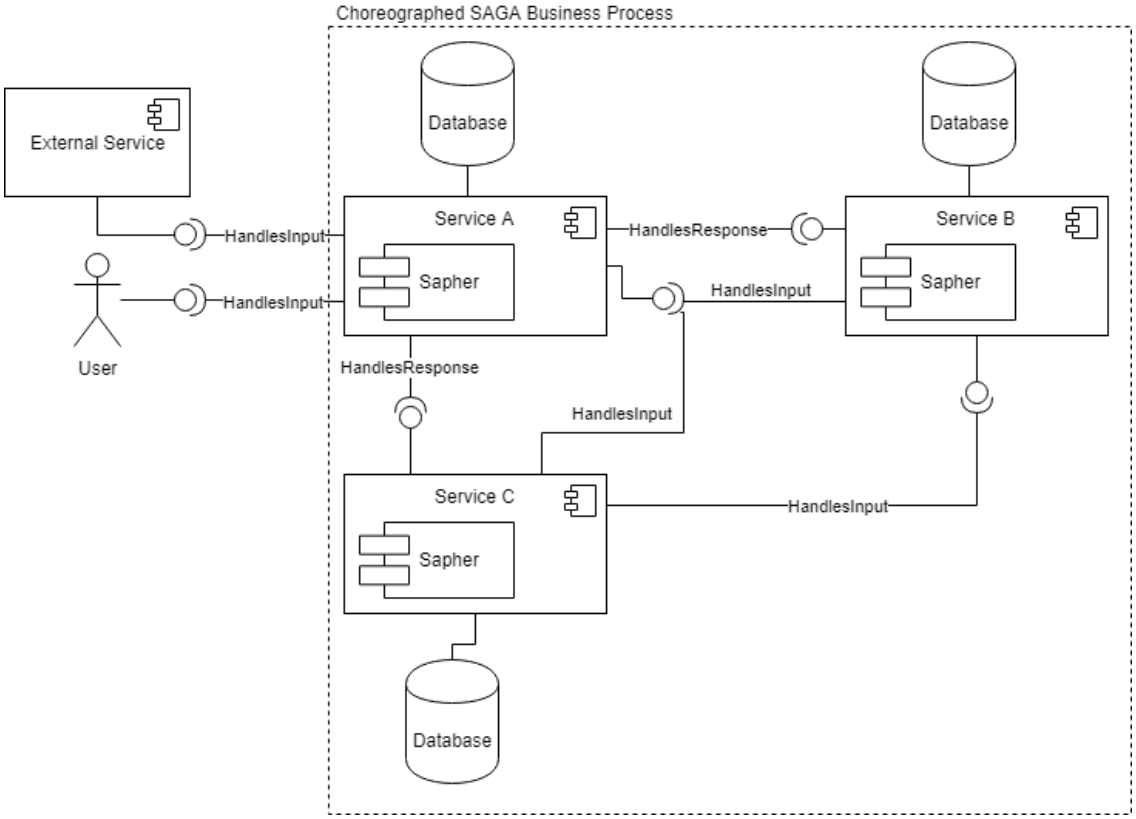


Figure 40 - Sapher high-level design view

As defined in Section 7.1.2 following the defined domain model, each business process step is executed in a different service, which respects the choreography approach. Sapher provides a generic implementation of these steps, which are, from now on, entitled as *"Sapher steps".*

Therefore, each microservice belonging to a distributed transaction should install Sapher and define the Sapher steps in which they are an active participant. For this reason, a Sapher step must have an input handler and can have multiple response handlers. In Sapher, an input handler is called *"HandlesInput"* and a response handler *"HandlesResponse".*

Figure 40 illustrates this in an example scenario constituted by three microservices, A, B, and C. Service A contains two SapherSteps, one triggered by a user action and another by an external system. Service B and C listen to the output of Service A triggering their own SapherSteps and publishing messages that service A can listen to as responses, using response handlers, and marking the business process step as successful. Finally, service C also listens to service B output to mark its own SapherStep as successful. Service C does not provide any output, as it is the final step of the business process. In each SapherStep, state management and persistence, idempotency, retries, and timeouts are managed by Sapher entirely.

All of these mechanisms and functionalities will be further detailed in the following sections.

7.2.2 Implementation view

This section describes some details of Sapher implementation: Sapher and SapherSteps modular configuration, handling mechanisms, and extensibility possibilities.

7.2.2.1 Configuration

Sapher is implemented in C#, using .NET technologies. Therefore, it uses the standard approach of .NET libraries and extends .NET *ServiceCollection*, using Microsoft Dependency Injection to define Sapher configuration details.

To do this, ServiceCollectionExtensions class provides an AddSapher method which allows the user to define Sapher configurations by accessing SapherConfigurator interface. The implementation allows the user to define logging, persistence, timeout and retry policies configuration. Also, SapherConfigurator uses SapherStepConfigurator to define SapherSteps, including its name, input handler, and response handlers. These handlers can be reused in different steps, and a message can have multiple handlers in different SapherSteps. Sapher will deliver the messages to all the SapherSteps that have any handler for the received message.

The method UseSapher of ServiceCollectionExtensions uses the generated SapherConfiguration and SapherStepConfiguration instances to create Sapher and SapherStep instances following the user configurations. This mechanism is illustrated in Figure 41.

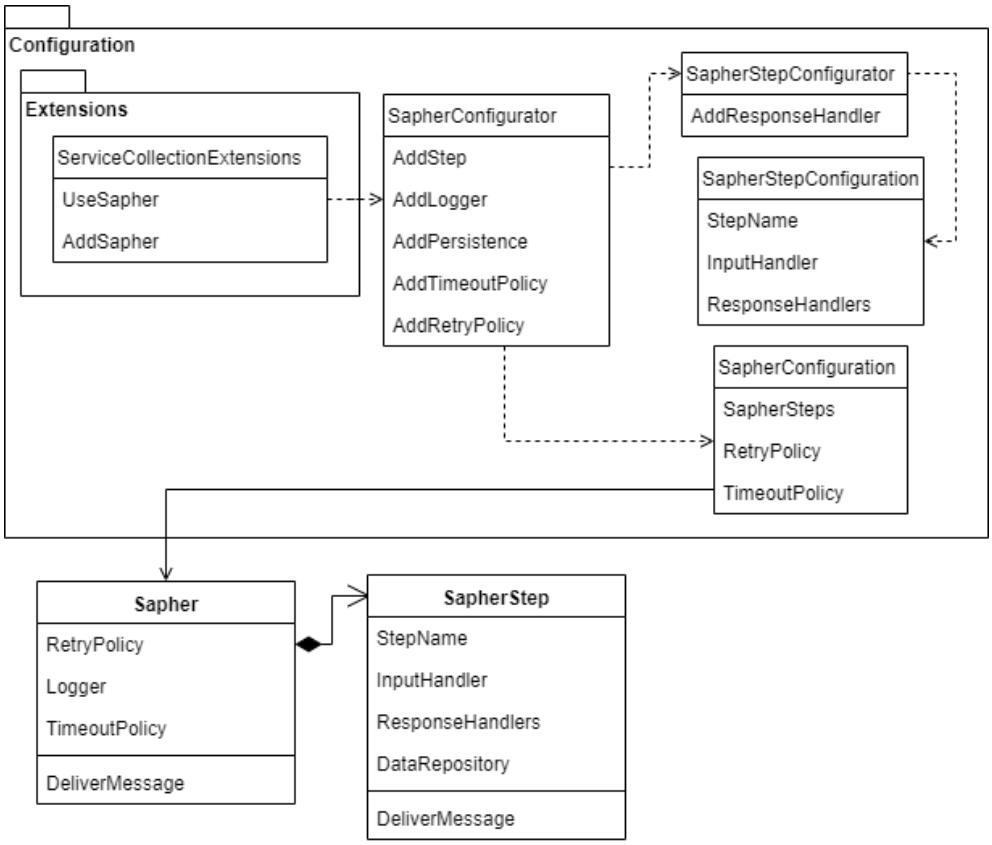


Figure 41 - Sapher configuration implementation view

7.2.2.2 Handlers

SapherStepConfigurator only accepts valid handlers. A valid handler is a class which implements the generic interfaces HandlesInput or HandlesResponse for a specific message type. Sapher uses these interfaces in the mediator pattern applied for message delivery - Figure 42.

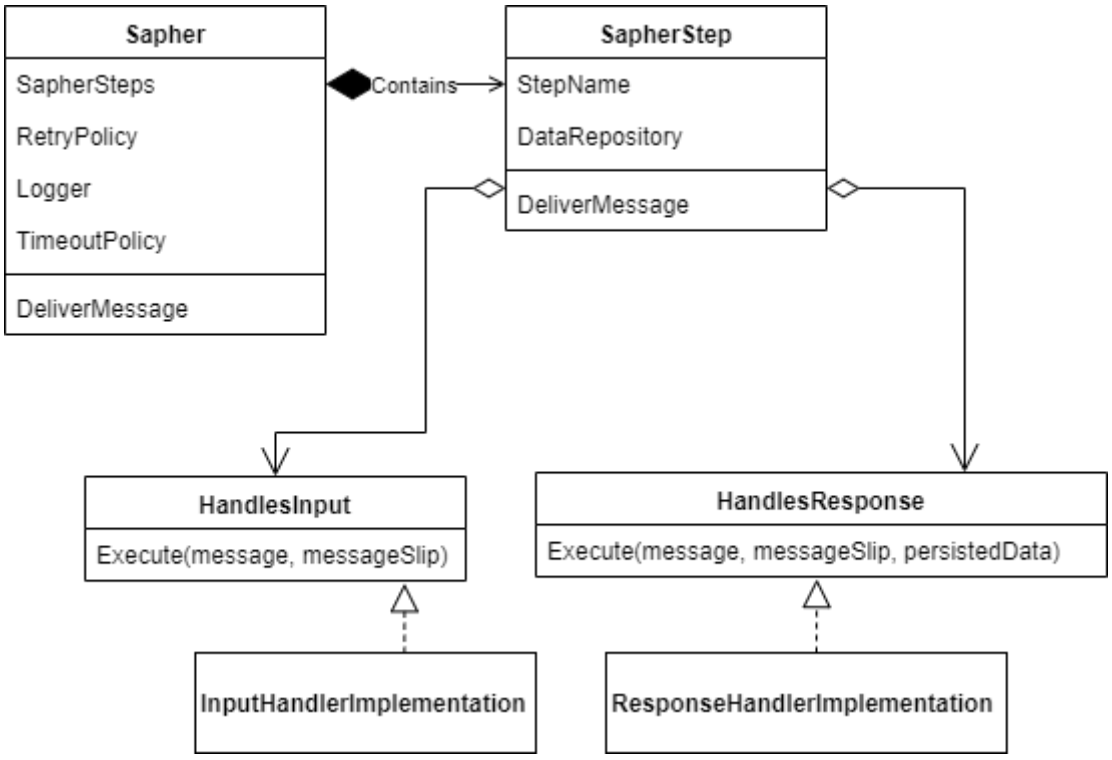


Figure 42 - Sapher handlers mediation

Sapher is agnostic to communication channels by providing a public method for message delivery. Users can receive messages by http, consume them from a message broker of any technology or any other communication channel, as long as they deliver the messages to Sapher delivery method, along with the message's message slip. Sapher acts as a mediator in message delivery by identifying the steps that handle that message and delivering it to the specific handler implementation configured in the AddSapher method previously described in this section. If the delivery fails, Sapher follows the defined retry policy to try again to deliver the message successfully.

Along with the mediation, Sapher ensures idempotency by persisting the message identifier together with the business process step state. Therefore, the same message is never consumed twice. More details can be seen in Figure 43.

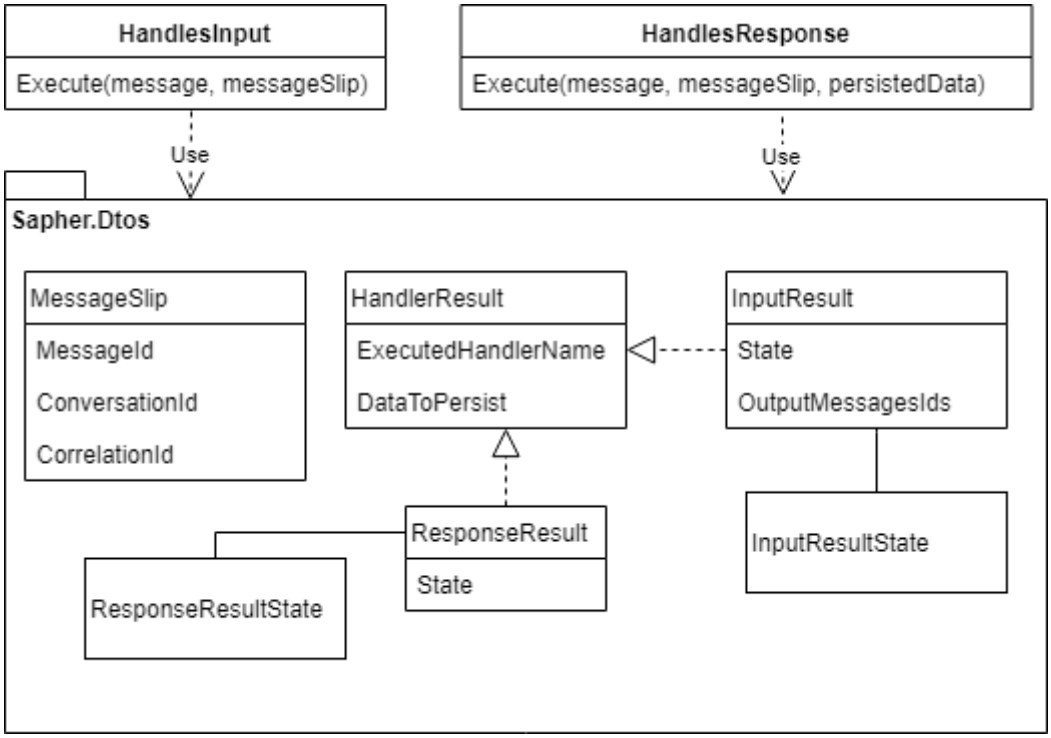


Figure 43 - Sapher execution state handling

HandlesInput and HandlesResponse implementations must provide an InputResult and ResponseResult, respectively. These data transfer objects (DTOs) are what allows Sapher to persist the state of the business process implemented by those handlers.

A response message can only be handled after the step has been instantiated by an input message. For that reason, InputResult contains the identifiers of the messages that were the output of the input message execution, if there were any. Response messages are correlated with these identifiers and therefore, the correspondent step state updated.

Both results allow the user to persist any valuable data regarding the execution. HandlesResponse implementations receive the previously persisted data so that they can coordinate the action to take, which can be useful to apply compensation actions.

If a SapherStep is instantiated by an input message where the InputResult contains output messages, Sapher will apply the user-defined timeout policy to mark the business process step as failed after a specified amount of time.

7.2.2.3 Logger

Sapher provides monitorization through logs of what is happening in the service, regarding message handling, step state persistence, among other useful information. To do that, Sapher provides a Logger interface and a LogEntry DTO, which can be implemented and extended respectively by the user. Therefore, developers using Sapher can keep using any logging framework that their microservice uses, or use a specific one for Sapher, as they prefer. If this logging implementation is not defined, Sapher will not log any message. This can be seen in Figure 44.

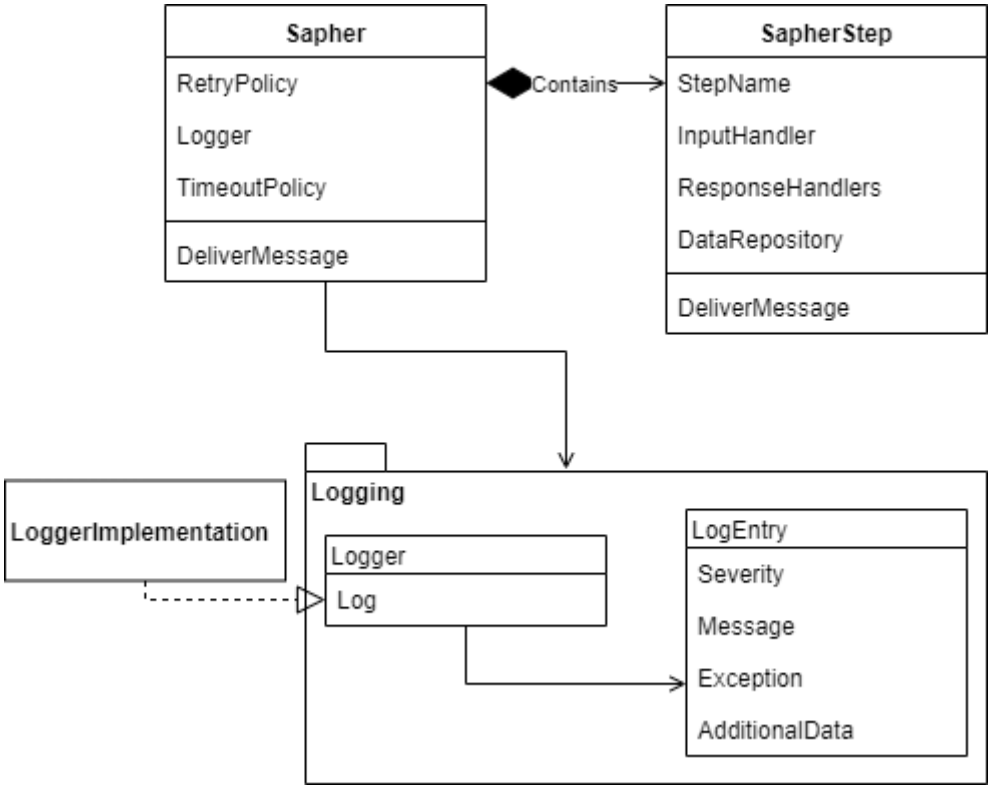


Figure 44 - Sapher logger extensibility

7.2.2.4 Persistence

Sapher persistence follows a similar approach to logging extensibility mechanism. Using the repository pattern, Sapher provides an interface SapherDataRepository that users can implement to use any storage engine as they see fit. Developers using Sapher also need to implement the adapter pattern in order to map their data model to Sapher DTOs. If an implementation of the repository is not provided, Sapher will use in-memory persistence to keep the state of the transactions, which is not recommended for distributed production environments.

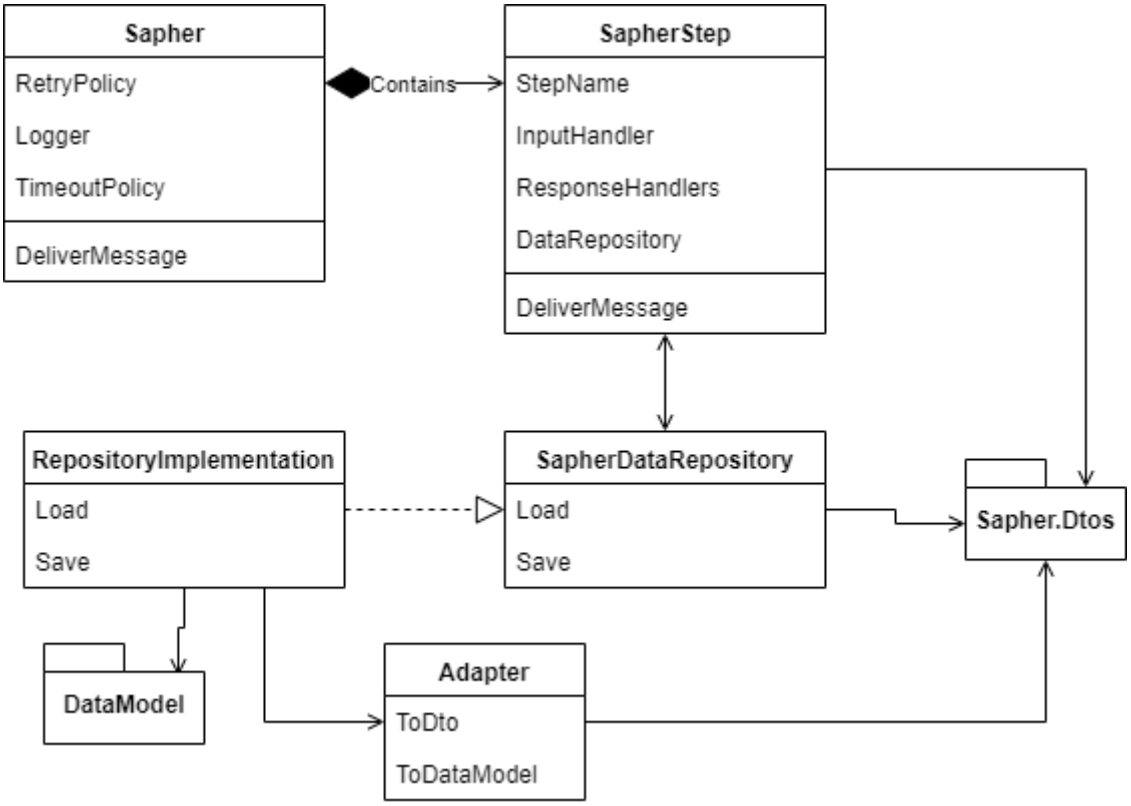


Figure 45 - Sapher persistence extensibility

7.2.3 Use cases specification

This section provides specific implementation details for each one of the defined use cases, ensuring that all of them are supported.

The definition of the business process step, logging extensions, and persistence configuration are made in Sapher setup process. Therefore, the correspondents UCs of these features (UC1, UC7, and UC8) can be visualised in Figure 46.

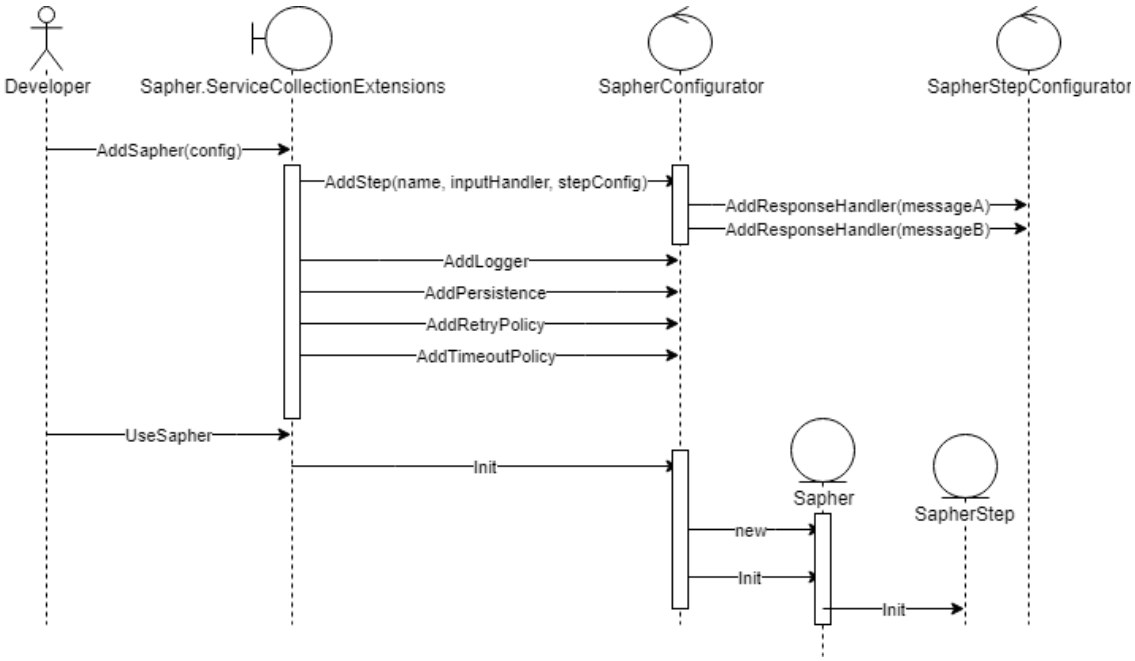


Figure 46 - Sapher configuration sequence diagram

When defining Sapher initial configuration, in the project setup, developers can define the steps and their respective inputs and responses, and reuse them in different steps.

Regarding UC2, the users can see the state of execution of the steps by requesting this information to Sapher, as can be seen in Figure 47.

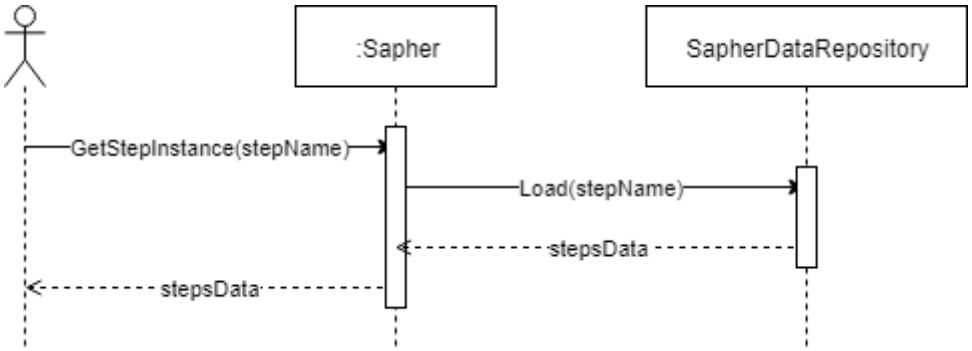


Figure 47 - Sapher state load sequence

In UC3, developers require the configuration of compensation actions, which can be accomplished by the implementation of HandlesResponse interface. A SapherStep that has a response handler (HandlesResponse) assigned to a message will execute the implementation of the mentioned interface. To facilitate the execution of compensation action, SapherStep provides the data persisted in the step input execution stage. Therefore, developers can use this information to apply compensation actions when receiving messages that require to do so. The functionality can be analysed in Figure 48.

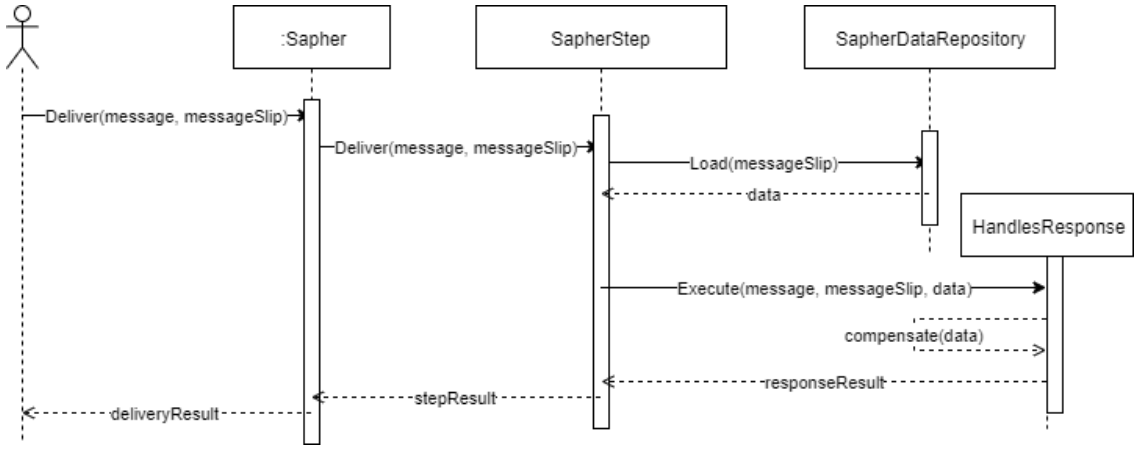


Figure 48 - Sapher compensation actions

To improve the resiliency of the system, UC4 requires the possibility of defining retry policies. The retry policy is configured in the setup of Sapher, as described in Figure 46. However, it is executed while delivering a message. When an exception is thrown during the delivery sequence, Sapher will follow the retry policy. Therefore, if retries are enabled, Sapher will retry the delivery the number of times specified, waiting for the specified time interval between each try. The functionality can be observed in Figure 49.

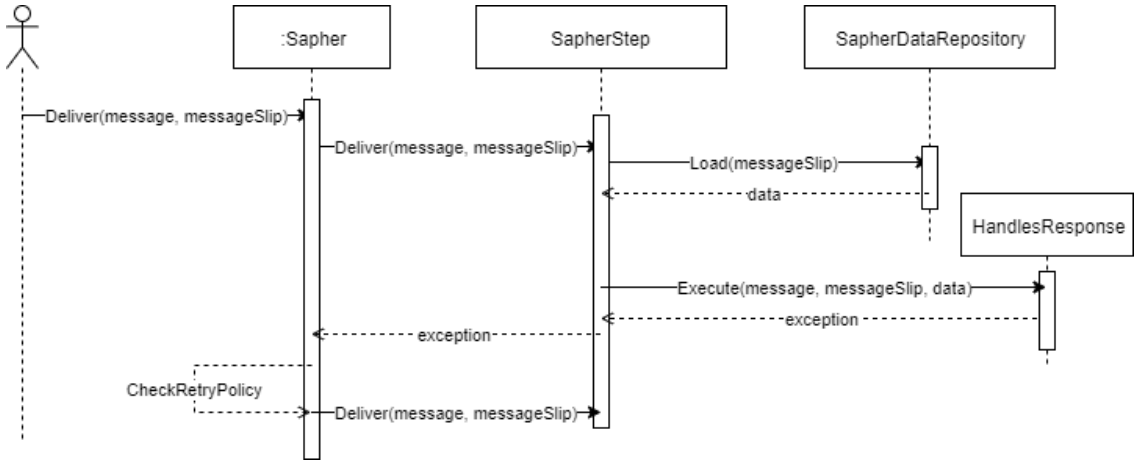


Figure 49 - Sapher retry execution

When using retries, the microservices need to be idempotent in order to avoid duplicated executions. Therefore, UC5 requires Sapher to implement idempotency in message consumers. SapherStep persists the identifier of all the messages it handles to accomplish that feature. When receiving a message, SapherStep verifies if the message was already consumed, and ignores it if it was already processed successfully. The process is illustrated in Figure 50.

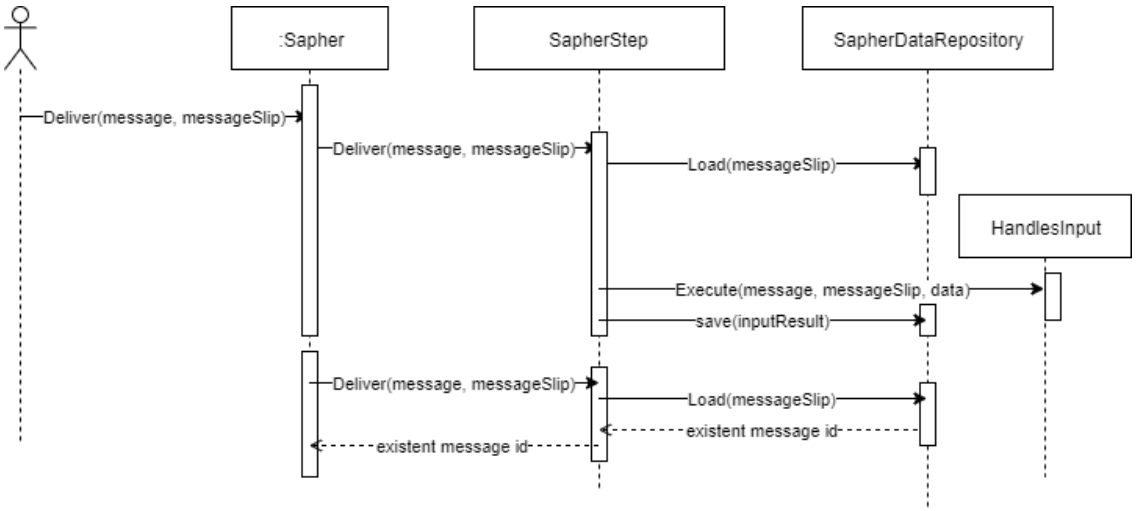


Figure 50 - Sapher idempotency

Finally, UC6 requires the possibility of defining timeout intervals for asynchronous request and reply executions. The timeout policy is configured in the setup of Sapher, as described in Figure 46. When a timeout policy is defined, Sapher will wait the specified amount for a response to a SapherStep execution. If a response takes longer than the defined period to reach Sapher, the state of the transaction will be marked as failed. The feature is detailed in Figure 51.

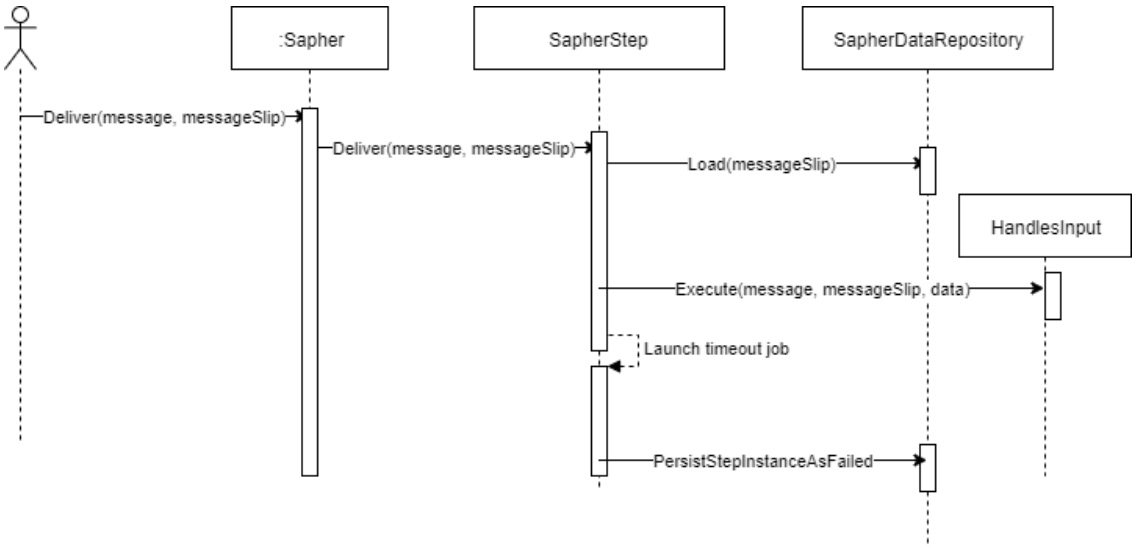


Figure 51 - Sapher timeout policy execution

7.2.4 Implementation process

To implement Sapher, Git was used as a version control system, hosting a C# repository on Github (https://github.com/joaodiasneves/sapher/) as it is the most common platform for open-source projects, with a large community which may be interested in supporting and contributing to Sapher in the future.

Also, Sapher was released in Nuget packages repository, the central package repository for .NET technologies. Continuous Integration and Continuous Delivery practices were used, using Travis-CI (https://travis-ci.com/joaodiasneves/sapher) as an integration and delivery pipeline, and semantic versioning (Preston-Werner, 2019) for project versioning.

During the development stage, the Test-Driven-Development was used. The first code to be written were unit tests, using the FluentAssertions third-party library. Then, Sapher started to be built around those tests until all the defined requirements passed.

The core features like message delivery through mediation and state management were the first to be implemented. After that, resiliency through idempotency, retries and timeouts were added using Polly open-source library (*App-vNext/Polly*, 2019). Then, extension points were defined for logging and persistence. Finally, a simple service was implemented using Sapher in order to have a proof of concept and usage sample of the developed library.

8 Evaluation

This chapter has the objective of evaluating the solutions implemented. It defines the metrics to be used, the hypotheses to be tested, the test methodology and the results of the tests. It also describes the approach used to evaluate the solutions.

In addition to the hypotheses tests, the results were also analysed using exploratory data analysis with mean, mode and median and using graphics to illustrate the data analysis.

8.1 Work validation by experts of the field

To evaluate this work, its results were introduced to experts of the microservices field who then provided their evaluation based on a questionnaire. The opinion of these professionals is of high importance as they have extensive experience in microservices migrations and implementations. The group of experts selected consists of Principal Engineers, Software Architects, among other roles that require extensive technical knowledge. The complete questionnaire can be found in appendix B.

The results introduced to the experts are divided into two sections:

1. Regarding the microservices migration research, the experts had access to section 6.5, which contains a summary of the results.
2. To evaluate the distributed transactions solution, the experts had access to the public repository of the project and its documentation.

The questionnaire provided contained three question groups:

1. Questions regarding the challenges identified.
2. Questions regarding solutions and best practices for each one of those challenges.
3. Questions regarding the distributed transactions solution.

All the questions are closed-ended, and the answers can be provided using values of the Likert scale (Likert, 1932).

Table 25 - Likert scale

Strongly Disapprove	Disapprove	Undecided	Approve	Strongly Approve
1	2	3	4	5

In order to analyse the results of the questionnaire regarding the groups of questions 1 and 2, each one will have specified intervals which will be described to the professionals before they answer the survey.

The intervals defined for the answers to group 1 are presented in Table 26 below.

Table 26 - Mean intervals for the evaluation of the problems identified

Interval	Description
[1-2]	The identified issues have no relation to microservices architecture or migration processes. The study does not bring value to the field.
[2-3]	Some of the issues are related to the microservices architecture or migration processes, but there are essential issues missing. The study does not bring value to the field.
[3-4]	The list of problems is complete and clear. The issues are related to the microservices architecture or migration processes, but some are not currently relevant. The study brings value to the field.
[4-5]	The study identified the most common challenges of microservices architecture and migration. The study brings value to the area.

The intervals defined for the answers to group 2 are presented in Table 27 below.

Table 27 - Mean intervals for the evaluation of the solutions and patterns identified

Interval	Description
[1-2]	The solutions and patterns identified have no relation to microservices architecture or migration processes. The study does not bring value to the field.
[2-3]	Some of the solutions and patterns identified are related to the microservices architecture or migration processes, but there are important techniques missing. The study does not bring value to the field.
[3-4]	The list of solutions and patterns is complete and clear. The methods are related to the microservices architecture or migration processes, but some are not currently relevant. The study brings value to the field.
[4-5]	The study identified the currently mostly used and proper techniques of microservices architecture and migration. The study brings value to the field.

Regarding group 3 (distributed transactions solution), the experts had access to the non-functional requirements and functional requirements. They were then asked to analyse the solution and provide feedback regarding the achievement of the requirements. To do that, they used the Likert scale to inform their evaluation from “Not achieved at all” (grade 1) to “achieved completely” (grade 5), for each group of requirements.

8.1.1 Preparation

The mean answer to each one of the groups was calculated and mapped to its specific interval. The description of the defined intervals gives some insights regarding the results of the group in particular.

In order to also have an overall evaluation, the mean of all the question groups was also calculated. This final grade will be used to test the value of this work.

Using the Likert scale, if a value is bigger than 3, then it is on the positive side of the range. Consequently, it is possible to consider that the work is valuable if the final mean is higher than 3. Therefore, we must understand if the final mean value is on the positive side of the scale, for which a One-Tailed t-Test was chosen, to test the following hypotheses.

$$H_0: Experts\ of\ the\ field\ consider\ this\ work\ results\ not\ valuable\ to\ the\ field$$
$$H_0: \mu \leq 3$$

$$H_1: Experts\ of\ the\ field\ consider\ this\ work\ results\ valuable\ to\ the\ field$$
$$H_1: \mu > 3$$

If the mean is more significant than 3, H_0 is refuted and therefore it is valid to say that the work results are valuable to the field.

8.1.2 Evaluation

The questionnaire was not publicly delivered. It was directly provided to specific professionals with recognized extensive knowledge in the microservices field. The main objective of this was to obtain feedback directly from experts and not from a wide group of professionals. It was possible to get 10 participants.

8.1.2.1 Experts background

Figure 52 illustrates the job titles of the participants. All of them require extensive technical knowledge and software architecture experience.

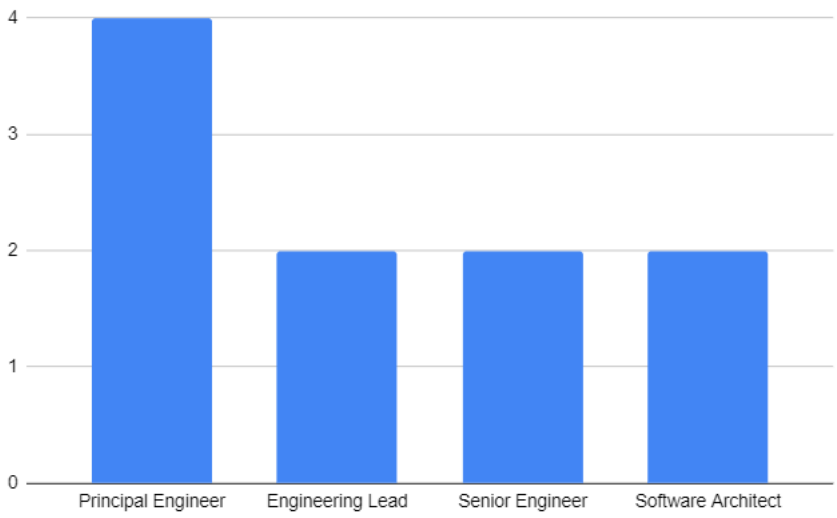


Figure 52 - Research validation - Participants job titles

Also, the participants answered how many years of experience they have, which can be analysed in Table 28.

Table 28 - Research validation - Participants years of experience

Participant	Years of experience
1	6
2	6
3	8
4	9
5	10
6	11
7	12
8	16
9	19
10	20
Average	11.7

All of the participants have at least 6 years of experience. The most experienced participant has 20 years of experience. Also, the average experience of the 10 participants is 11.7. Therefore, the participants are highly experienced and are able to provide value by evaluating this work results.

8.1.2.2 Main challenges study evaluation

After gathering some information regarding the participant's profiles, they were asked to provide feedback regarding the main challenges identified (questions group 1).

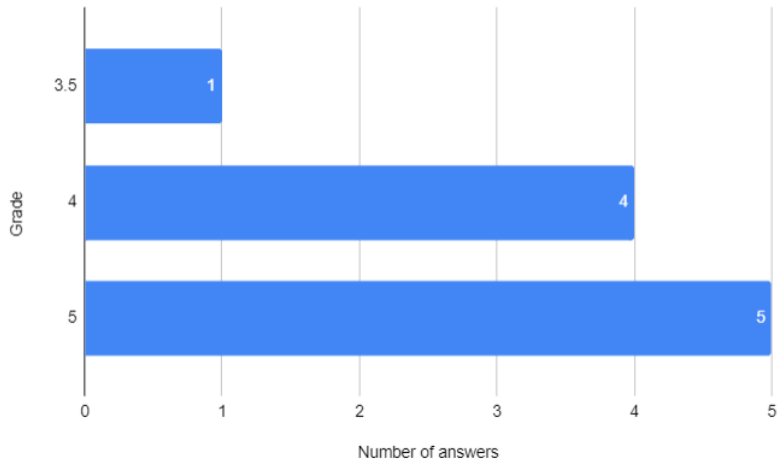


Figure 53 - Research validation - Main challenges grade

Figure 53 describes the grades provided by experts regarding the main challenges identified in the study. Half of the experts provided the maximum grade of 5, while the lowest grade was 3.5. Therefore, all of the experts provided a positive evaluation (more than 3 in the Likert scale). The mean grade of all participants was 4.45, positioning the main challenges evaluation in the following descriptive grade interval:

- [4-5] - "The study identified the most common challenges of microservices architecture and migration. The study brings value to the area."

Two of the experts provided additional comments:

1. "There are some technical challenges like the local developer experience and resiliency that are uncovered during this research."
2. "I would consider resiliency aspects of such architecture as the main challenge."

While the local developer experience was not given a close attention in this work because it is not a technical challenge, resiliency is identified as an improvement point for future work.

8.1.2.3 Solutions and best practices evaluation

For each one of the challenges, some solutions and best practices were identified. Therefore, the experts were asked to evaluate them following the Likert scale. The results can be observed in Figure 54.

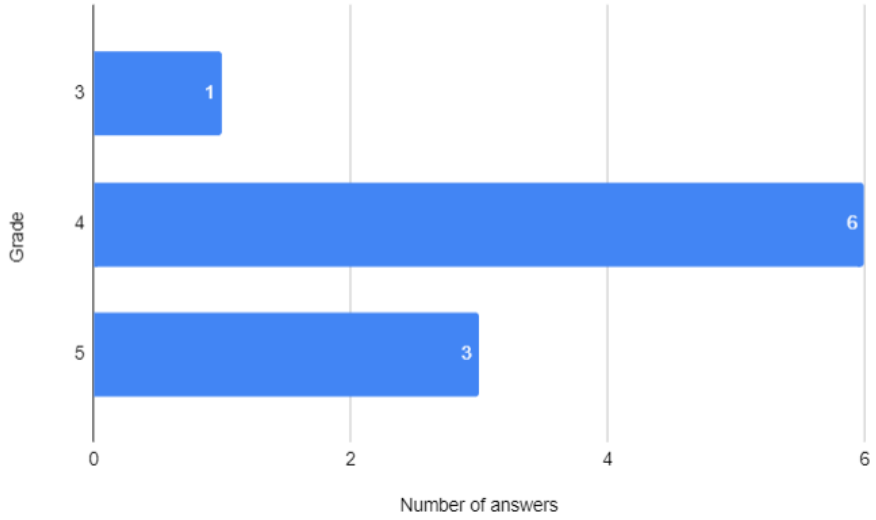


Figure 54 - Research validation – Solutions and best practices grade

Most of the participants classified the solutions and best practices in grade 4. The lowest grade was 3 and the highest grade was 5. These values have a mean of 4.2, which is lower than the grade of the main challenges but is still on the positive side of the Likert scale. Also, 4.2 is in the interval between 4 and 5 providing the following descriptive grade:

- [4-5] – "The study identified the currently mostly used and proper techniques of microservices architecture and migration. The study brings value to the field."

The expert that classified the solutions at grade 3 justified it stating the following comment:

- "In order to have better visibility of a distributed system besides logging and tracing, other practices are important on that context as metrics aggregators and alerting".

In fact, the solutions to the distributed monitoring challenge do not mention metrics or alerting, constituting, therefore, a gap to be pursued in future work.

8.1.2.4 Distributed transactions solution evaluation

After evaluation of the microservices migration research, participants were asked to evaluate the distributed transaction solution by accessing the code in the public repository along with its documentation. The experts evaluated the requirements of the project by providing a grade from 1 to 5, where 1 means that the requirements were not met and 5 that the requirements were all achieved.

Figure 55 provides the answers regarding non-functional requirements.

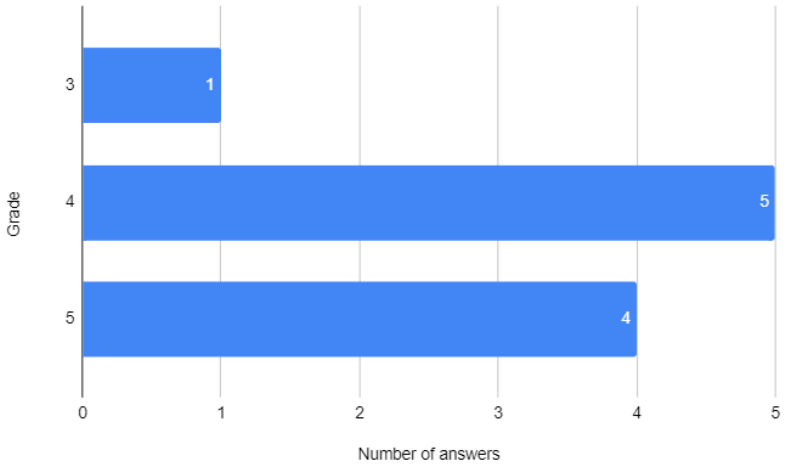


Figure 55 - Distributed transactions solution evaluation – non-functional requirements

While only one expert provided grade 3, all the others provided higher grades, positioning the solution with a mean of 4.3, which is on the positive side of Likert scale.

Regarding the functional requirements, the same approach was used, and the results can be found in Figure 56.

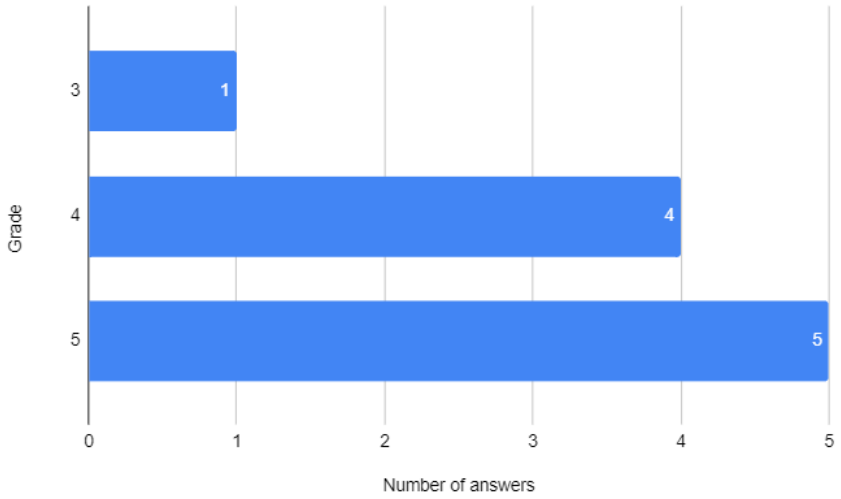


Figure 56 - Distributed transactions solution evaluation - functional requirements evaluation

While half of the participants evaluated the non-functional requirements in grade 4, in the case of functional requirements 50% of the experts rated the solution at grade 5. However, one of the participants evaluated the solution with value 3. This provides a mean grade of 4.4 for functional requirements, only 0.1 points of difference from the non-functional.

Finally, two of the participants provided some comments regarding the solution:

- “Amazing work and analyses with real-life use cases and experience.”
- “Excellent work, thanks for sharing it.”

To test the hypotheses of this evaluation, the total mean of question group 3 (distributed transactions solution) must be calculated. The results can be found in Table 29.

Table 29 – Distributed transactions solution evaluation – Means

Description	Mean grade
Non-functional requirements	4.3
Functional requirements	4.4
Total mean	4.35

8.1.2.5 Hypotheses test

To conclude this evaluation, the hypotheses stated in section 8.1.1 must be tested. As previously mentioned, this will be obtained by calculating the total mean of the answers and positioning it on the Likert scale. The calculations are present in Table 30.

Table 30 - Work evaluation - total means

Question group	Mean grade
1 – Main challenges	4.45
2 – Solutions and best practices	4.2
3 – Distributed transactions solution	4.35
Total mean	4.3

The total mean of this evaluation is 4.3, which is more significant than 3, positioning the evaluation in the positive side of the Likert scale.

$$\mu = 4.3$$

$$4.3 \geq 3$$

With this value, H_0 is refuted and therefore it is valid to say that the work results are valuable to the field.

9 Conclusions

This chapter concludes this document by analysing and comparing the initially defined objectives with the work outputs and outcomes. The difficulties identified during this work are described here, along with possible future work.

9.1 Achieved objectives

In Section 5.2 the main objectives of this work were defined. In this section, the achievements of these objectives are evaluated and justified with corresponding evidence. Table 31 presents the different objectives and their completeness.

Table 31 - Objectives achievement

Number	Objective	Completeness
1	Identify the most common technical challenges that teams currently face while migrating to the microservices architecture and possible solutions.	Achieved
2	Address the distributed transactions challenge specifically, proposing a solution to ease the management of distributed transactions in a microservices architecture, using choreographed sagas.	Achieved

This work contributed to the microservices field with a catalogue of the most common challenges faced by teams when adopting the microservices architecture. Also, for each one of these challenges, some solutions were identified. The results were obtained by conducting a systematic mapping study analysing 54 different articles published since 2018. Also, an industry survey was completed by 30 industry professionals with experience in microservices architectures. Furthermore, a participant observation study of a real microservices migration was conducted. Therefore, it is possible to conclude that objective 1 was achieved successful, and the evidence can be found in Chapter 6.

Regarding objective 2, different strategies to solve the distributed transactions challenge were analysed. After, a solution was implemented using the choreographed saga pattern. The project was publicly published as open-source, and all the details can be found in Chapter 7, containing the evidence necessary to consider objective 2 as achieved.

In Chapter 8, 10 experts in microservices with an average of 11.7 years of experience in the field provided a positive grade (4.3) in the Likert Scale (1 to 5) and feedback regarding both the microservices research study and the distributed transactions challenge, considering the work valuable to the field, and providing further evidence of their achievement.

9.2 Difficulties along the way

During the development of this work, different difficulties were faced influencing the final results of this work:

- Industry survey too long – one of the difficulties found was to find enough participants in the industry survey. Some participants mentioned that they took a long time to answer the questionnaire even though the questions were close-ended, due to the number of questions. This limitation may have reduced the number of participants, but obtained more information from each participant.
- Confidentiality issues – due to confidentiality issues it was not possible to provide all the obtained information from the participant observation study.
- Experts availability – another difficulty found during this work was to find availability from industry experts to validate the results of this work as it required them to read the results and analyse the distributed transactions solution implemented in detail, which demands some time.
- Companies interest – it was challenging to find companies available to test the developed distributed transactions solution in a production environment. The initially agreed company changed its priorities and refused to implement the solution in the timeframe required by this work.
- Initial idea concretization – initially this work objective was to define a guide for migration of monoliths to the microservices architecture. However, during the context and state of the art analysis, a different path for this work was chosen in order to provide more tangible value, which also caused some changes in the initial project structure.

9.3 Future work

Even though this work achieved its objectives, there are always improvement points. Also, this work's contributions identify essential challenges for further research in the microservices field.

One of the areas mentioned by professional experts in Section 8.1.2 that should be pursued in future work is resiliency, specifically alerting and metrics aggregation.

Also, this work focuses on technical challenges, however, multiple organizational and cultural challenges were found in the process of migrating to the microservices architecture, and are therefore areas left open for future work – namely organization and team structure to implement microservices, team members experience, deliveries planning and the software development lifecycle, specifically the possible testing stages.

Regarding the distributed transactions solution, future work constitutes the implementation of the solution in a production environment in order to validate its viability in a real usage scenario. The preparation of this experiment is detailed in appendix C and can be executed in the future, as the projected is published as open-source.

Furthermore, logging and persistence implementations can be developed using the extensibility points provided by Sapher in order to facilitate the usage of the solution by interested teams or individuals. Other features, such as alert and caching can be implemented in the future.

Additionally, Sapher was designed in line with the choreographed saga pattern to provide an implementation for this approach as in Section 4.3.2 the analysed solutions-focused only in the orchestration style. However, even though choreography and orchestration were compared describing the multiple advantages and disadvantages of each, the use cases for using one or the other were not researched, leaving a gap for future research, which would also provide more guidance for the right use cases for Sapher usage.

Finally, this work was written in English so that it can reach a higher number of readers. Also, it was structured with the possibility of publishing an article on a recognised platform or conference, further increasing the work reach. Even though it was not possible to achieve this in the timeframe available for this work, this task will be completed in the future, so that a different perspective of reviews can be gathered, and future research influenced positively.

References

Amazon, 2019. AWS Step Functions [WWW Document]. URL https://aws.amazon.com/step-functions/ (accessed 8.10.19).

Aoyama, M., 1998, April. New age of software development: How component-based software engineering changes the way of software development. In 1998 International Workshop on CBSE (pp. 1-5).

App-vNext/Polly, 2019. . App vNext.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A. and Lynn, T., 2018. Microservices migration patterns. Software: Practice and Experience, 48(11), pp.2019-2042.

Baškarada, S., Nguyen, V. and Koronios, A., 2018. Architecting Microservices: Practical Opportunities and Challenges. Journal of Computer Information Systems, pp.1-9.

Bonham, A., 2019. Microservices — When to React Vs. Orchestrate [WWW Document]. Medium. URL https://medium.com/capital-one-tech/microservices-when-to-react-vs-orchestrate-c6b18308a14c (accessed 8.10.19).

Camunda, 2019. Workflow and Decision Automation Platform [WWW Document]. Camunda BPM. URL https://camunda.com/ (accessed 8.10.19).

Carrasco, A., Bladel, B.V. and Demeyer, S., 2018, September. Migrating towards microservices: migration and architecture smells. In Proceedings of the 2nd International Workshop on Refactoring (pp. 1-6). ACM.

Cerny, T., Donahoo, M.J. and Trnka, M., 2018. Contextual understanding of microservice architecture: current and future directions. ACM SIGAPP Applied Computing Review, 17(4), pp.29-45.

Chen, L., 2018, April. Microservices: architecting for continuous delivery and DevOps. In 2018 IEEE International Conference on Software Architecture (ICSA) (pp. 39-397). IEEE.

Ciavotta, M., Alge, M., Menato, S., Rovere, D. and Pedrazzoli, P., 2017. A microservice-based middleware for the digital factory. Procedia Manufacturing, 11, pp.931-938.

Di Francesco, P., Lago, P. and Malavolta, I., 2018, April. Migrating towards microservice architectures: an industrial survey. In 2018 IEEE International Conference on Software Architecture (ICSA) (pp. 29-2909). IEEE.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L., 2017. Microservices: yesterday, today, and tomorrow. In Present and ulterior software engineering (pp. 195-216). Springer, Cham.

Fowler, M., 2018. How to break a Monolith into Microservices [WWW Document]. martinfowler.com. URL https://martinfowler.com/articles/break-monolith-into-microservices.html (accessed 10.13.18).

Fowler, M., 2015a. Monolith First [WWW Document]. martinfowler.com. URL https://martinfowler.com/bliki/MonolithFirst.html (accessed 4.15.19).

Fowler, M., 2015b. MicroservicePremium [WWW Document]. martinfowler.com. URL https://martinfowler.com/bliki/MicroservicePremium.html (accessed 6.30.19).

Fowler, M., 2015c. bliki: MonolithFirst [WWW Document]. martinfowler.com. URL https://martinfowler.com/bliki/MonolithFirst.html (accessed 6.30.19).

Fowler, M., Lewis, J., 2014. Microservices [WWW Document]. martinfowler.com. URL https://martinfowler.com/articles/microservices.html (accessed 10.28.18).

Fox, A. and Brewer, E.A., 1999, March. Harvest, yield, and scalable tolerant systems. In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (pp. 174-178). IEEE.

Fritzsch, J., Bogner, J., Zimmermann, A. and Wagner, S., 2018, March. From monolith to microservices: a classification of refactoring approaches. In International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment (pp. 128-141). Springer, Cham.

Furda, A., Fidge, C., Zimmermann, O., Kelly, W. and Barros, A., 2017. Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency. IEEE Software, 35(3), pp.63-72.

Garcia-Molina, H. and Salem, K., 1987. Sagas (Vol. 16, No. 3, pp. 249-259). ACM.

Gerlag, D., 2019. Lightweight workflow engine for .NET Standard. Contribute to danielgerlag/workflow-core development by creating an account on GitHub.

Gilbert, S. and Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. Acm Sigact News, 33(2), pp.51-59.

Google, 2019. Google Forms About Section.

Gray, J. and Lamport, L., 2006. Consensus on transaction commit. ACM Transactions on Database Systems (TODS), 31(1), pp.133-160.

Gray, J. and Reuter, A., 1992. Transaction processing: concepts and techniques. Elsevier.

Greenfield, P., Fekete, A., Jang, J. and Kuo, D., 2003, September. Compensation is not enough [fault-handling and compensation mechanism]. In Seventh IEEE International Enterprise Distributed Object Computing Conference, 2003. Proceedings. (pp. 232-239). IEEE.

Hasselbring, W. and Steinacker, G., 2017, April. Microservice architectures for scalability, agility and reliability in e-commerce. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 243-246). IEEE.

Heidari, F., Loucopoulos, P., 2014. Quality evaluation framework (QEF): Modeling and evaluating quality of business processes. Int. J. Account. Inf. Syst. 15, 193–223. https://doi.org/10.1016/j.accinf.2013.09.002

Hohpe, G., Woolf, B., 2004. Enterprise Integration Patterns: Deisgning, Building, and Deploying Messaging Solutions. Addison-Wesley.

Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J. and Tilkov, S., 2018. Microservices: The journey so far and challenges ahead. IEEE Software, 35(3), pp.24-35.

Knoche, H. and Hasselbring, W., 2018. Using microservices for legacy software modernization. IEEE Software, 35(3), pp.44-49.

Koen, P.A., Ajamian, G.M., Boyce, S., Clamen, A., Fisher, E., Fountoulakis, S., Johnson, A., Puri, P. and Seibert, R., 2002. Fuzzy front end: effective methods, tools, and techniques. The PDMA toolbook 1 for new product development.

Kopp, O., Wieland, M., Leymann, F., 2009. Towards Choreography Transactions 7.

Kosaraju, M., 2007. XA transactions using Spring [WWW Document]. JavaWorld. URL https://www.javaworld.com/article/2077714/xa-transactions-using-spring.html (accessed 4.23.19).

Lake, B., 2012. An empirical evaluation of an agile modular software development approach: A case study with Ericsson.

Larrucea, X., Santamaria, I., Colomo-Palacios, R. and Ebert, C., 2018. Microservices. IEEE Software, 35(3), pp.96-100.

Lethbridge, T.C., Sim, S.E. and Singer, J., 2005. Studying software engineers: Data collection techniques for software field studies. Empirical software engineering, 10(3), pp.311-341.

Levcovitz, A., Terra, R., Valente, M.T., 2016. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. ArXiv160503175 Cs.

Likert, R., 1932. A technique for the measurement of attitudes. Archives of psychology.

MuleSoft, Inc, 2018. Best Practices for Microservices [WWW Document]. URL https://www.mulesoft.com/lp/whitepaper/api/microservices-bestpractices

Murphy-Hill, E. and Black, A.P., 2008. Refactoring tools: Fitness for purpose. IEEE software, 25(5), pp.38-44.

Narumoto, M., Wilson, M., Buck, A., Wasson, M., 2017. Strangler pattern - Cloud Design Patterns [WWW Document]. URL https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler (accessed 6.23.19).

Netflix, 2019. Introduction - Conductor [WWW Document]. URL https://netflix.github.io/conductor/ (accessed 8.10.19).

Newman, S., 2015. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.".

Ntentos, E., Zdun, U., Plakidas, K., Schall, D., Li, F. and Meixner, S., 2019. Supporting Architectural Decision Making on Data Management in Microservice Architectures.

Open Group, 1991. Distributed transaction processing: the XA specification. X/Open, Reading.

Pahl, C. and Jamshidi, P., 2016, April. Microservices: A Systematic Mapping Study. In CLOSER (1) (pp. 137-146).

Particular, 2019. Sagas • NServiceBus • Particular Docs [WWW Document]. URL https://docs.particular.net/nservicebus/sagas/ (accessed 8.7.19).

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J. and Josuttis, N., 2017. Microservices in practice, part 1: Reality check and service design. IEEE Software, 34(1), pp.91-98.

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.M., 2017b. Microservices in Practice, Part 2: Service Integration and Sustainability. IEEE Softw. 34, 97–104.

Preston-Werner, T., 2019. Semantic Versioning 2.0.0 [WWW Document]. Semantic Versioning. URL https://semver.org/ (accessed 9.22.19).

Ren, Z., Wang, W., Wu, G., Gao, C., Chen, W., Wei, J. and Huang, T., 2018, September. Migrating Web Applications from Monolithic Structure to Microservices Architecture. In Proceedings of the Tenth Asia-Pacific Symposium on Internetware (p. 7). ACM.

Rich, N. and Holweg, M., 2000. Value analysis. Value engineering: Innoregio: dissemination of innovation and knowledge management techniques, report produced for the EC funded project. United Kingdom: Lean Enterprise Research Centre Cardiff.

Richards, M., 2015. Microservices vs. service-oriented architecture.

Richardson, C., 2019. Eventuate [WWW Document]. URL https://eventuate.io/ (accessed 8.10.19).

Rosa, D., 2018a. Saga Pattern | How to implement business transactions using Microservices - Part I [WWW Document]. Couchbase Blog. URL https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/ (accessed 2.23.19).

Rosa, D., 2018b. Saga Pattern | How to implement business transactions using Microservices – Part II [WWW Document]. Couchbase Blog. URL https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2/ (accessed 2.23.19).

Saaya, Z., Mustafa, N. and Devaraju, A., 2007. The Development of Practical Guidelines for Designing Online Questionnaires.

Sampaio, A., 2015. Improving systematic mapping reviews. ACM SIGSOFT Software Engineering Notes, 40(6), pp.1-8.

Shier, R., 2004. Statistics: 1.1 Paired t-tests.

Stafford, J., 2018. How to avoid common challenges when migrating to microservices [WWW Document]. SearchMicroservices. URL https://searchmicroservices.techtarget.com/news/450427289/How-to-avoid-common-challenges-when-migrating-to-microservices (accessed 10.13.18).

Stricker, R., Müssig, D. and Lässig, J., 2018. Microservices for Redevelopment of Enterprise Information Systems and Business Processes Optimization. In ICEIS (2) (pp. 719-726).

Strîmbei, C., Dospinescu, O., Strainu, R.M. and Nistor, A., 2015. Software architectures–Present and visions. Informatica Economica, 19(4), p.13.

Taibi, D., Lenarduzzi, V. and Pahl, C., 2018, March. Architectural Patterns for Microservices: A Systematic Mapping Study. In CLOSER (pp. 221-232).

Torgerson, C., 2003. Systematic reviews. Bloomsbury Publishing.

Trihinas, D., Tryfonos, A., Dikaiakos, M.D. and Pallis, G., 2018. Devops as a service: Pushing the boundaries of microservice adoption. IEEE Internet Computing, 22(3), pp.65-71.

Uber Engineering, 2019. Welcome | Cadence [WWW Document]. URL https://cadenceworkflow.io/ (accessed 8.10.19).

Ulander, D., 2017. Software Architectural Metrics for the Scania Internet of Things Platform: From a Microservice Perspectiv.

Ulkhaq, M.M., Wijayanti, W.R., Zain, M.S., Baskara, E. and Leonita, W., 2018, March. Combining the AHP and TOPSIS to evaluate car selection. In Proceedings of the 2nd International Conference on High Performance Compilation, Computing and Communications (pp. 112-117). ACM.

Vogels, W., 2009. Eventually consistent. Communications of the ACM, 52(1), pp.40-44.

Vresk, T. and Čavrak, I., 2016, May. Architecture of an interoperable IoT platform based on microservices. In 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 1196-1201). IEEE.

Xiang, K., 2018. Patterns for distributed transactions within a microservices architecture. Red Hat Dev. Blog. URL https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/ (accessed 2.21.19).

Zimmermann, O., 2015. Architectural refactoring: A task-centric view on software evolution. IEEE Software, 32(2), pp.26-29.

Appendix A

Technical Challenges of Microservices Migration - Questionnaire

A study and possible development of a technical guide for migrating systems to Micro-Services oriented architectures are being created in the scope of a Master's Thesis on Software Engineering for the School of Engineering of Porto Polytechnic Institute (ISEP).

For this reason, there is the need to identify the common activities and challenges faced by professionals on the field when dealing with this kind of process.

Your google account log-in is required only to assure that you answer only once. The questionnaire is anonymous.
However, the study will provide solutions for the identified most common challenges. So at the end of the questions, you can subscribe to the study in order to be notified about its results if you are interested.

Please only answer if you have been involved in a system migration towards a microservices architecture, or are planning to do so in the near future.

Please think of a specific migration project you were involved in while answering the questionnaire.

The questions are divided into 5 different sections: Introduction, Existing system analysis, Designing the new architecture, implementing the new architecture and a section for any additional notes.

It is expected that you take a maximum of 30 minutes to complete this questionnaire.

Thank you for your time. Please share this questionnaire with your contacts. It is really important for the study.

Introduction

This section has the purpose of gathering some information about you and your migration experience

1. **How many years of professional experience in the software field do you have?** *

_____

2. **What was your role at the time of the migration?** *
   Mark only one oval.

   - ◯ Software Architect
   - ◯ CTO
   - ◯ Software Engineer
   - ◯ Infrastructure Engineer
   - ◯ Quality Assurance Engineer
   - ◯ Project Manager
   - ◯ Other: _____

3. **Please select the sentence that best describes your system before the migration** *
Mark only one oval.

- ◯ My system is a single Monolithic application (a software system whose modules cannot be executed independently)
- ◯ My system consists of a few Monolithic applications (2 to 5).
- ◯ My system consists of more than 5 services , but some of them are Monolithic
- ◯ Other: _____

4. **How many services did/do you have before the migration?** *

_____

5. **In what stage of the migration are you right now?** *
Mark only one oval.

- ◯ Early Stage - either only planned or just started
- ◯ 30% - 50% of progress
- ◯ 70% - 80% of progress
- ◯ Fully completed
- ◯ Other: _____

6. **How many months did you expect it would take to complete the migration?** *

_____

7. **How many months did it actually take to complete the migration?** *

_____

8. **Why did your company decide to migrate? Select up to 4 reasons that you consider most important** *

Tick all that apply.

- [ ] Long time to release new features
- [ ] High coupling
- [ ] Unpredictable side effects while developing new features.
- [ ] Low productivity of developers
- [ ] Unknown boundaries of functionalities or modules
- [ ] Poor confidence on the source code
- [ ] Making information explicit
- [ ] Performance and/or scalability issues
- [ ] The existing system was hard to test
- [ ] I don't know/It was not my decision
- [ ] Because everybody does it
- [ ] Fault tolerance
- [ ] Easy technology experimentation
- [ ] Delegation of software responsibilities
- [ ] Other: _____

9. **Please describe the different steps taken in the migration process. Example: (Pre-existing system analysis, new system design, new services implementation prioritization, implementation, testing, etc...)**

_____

_____

_____

_____

Existing system analysis

The objective of this session is to understand if you analyzed the existing system while planning the migration and how did you do it.

10. **What were the sources used to analyze the existing system? Select all that apply** *
Tick all that apply.

- [ ] I did not analyze the existing system when planning the migration
- [ ] Source code
- [ ] Textual documents
- [ ] Tests
- [ ] Meetings
- [ ] Data models/schema
- [ ] Unwritten knowledge among developers and engineers
- [ ] Presentations
- [ ] Diagrams (UML or other)
- [ ] Contracts with customers
- [ ] Information extracted from architecture recovery tools
- [ ] Other: _____

11. **Why did you analyze the existing system? Select all that apply** *
Tick all that apply.

- [ ] I did not analyze the existing system when planning the migration
- [ ] Understanding the existing system
- [ ] Architecting the new system
- [ ] Defining processes and APIs in the new system
- [ ] Finding dependencies in the existing system
- [ ] Creating tests for the new system
- [ ] Other: _____

12. **What were the main challenges faced while analyzing the existing system? Select up to 4 reasons that you consider most important** *
Tick all that apply.

- [ ] Missing architectural documentation
- [ ] Code had no comments
- [ ] Non-existent or reduced functional specification
- [ ] Non-existent or reduced test cases specification
- [ ] Technologies of the system were very old
- [ ] The system was so old that no one on the company actually knew it entirely
- [ ] System APIs had no specification or documentation
- [ ] The system and/or functionalities dependencies were not documented or identified
- [ ] Databases schema documentation were outdated or non-existent
- [ ] Other: _____

13. **Please provide any additional information you find useful regarding the existing system analysis**

Designing the new architecture

This section intends to understand the process used to design the new system.

14. **Which of the following activities did you perform while designing the new system? Select all that apply** *

Tick all that apply.

- [ ] Domain decomposition
- [ ] Identification of the services for the new system
- [ ] Application of Domain-Driven design practices
- [ ] System decomposition
- [ ] Building proof-of-concept services to assess feasibility of the migration
- [ ] Boundary identification in the pre-existing system
- [ ] Implementing services as Minimum Viable Products
- [ ] Dependencies identification in the existing system
- [ ] Careful design of the business workflows
- [ ] Reduce risks in the network layer
- [ ] Other:

15. **How did you document the design of the new architecture? Select all that apply** *

Tick all that apply.

- [ ] Architectural documents
- [ ] Textual documents
- [ ] Diagrams (UML and others)
- [ ] Presentations
- [ ] Source code with annotations
- [ ] Notes
- [ ] Domain-specific language models
- [ ] Video
- [ ] I did not document it
- [ ] Other:

16. **What was the driver while designing the new system?** *

Mark only one oval.

- ◯ The functionalities (migrate new functionalities before migrating the existing ones or migrate the most important functionalities first - being them new or old)
- ◯ The type of customer of the system (migrate according to the customer's inputs and needs)
- ◯ Management/business (migrate according to the specification given by the management/business - regardless of their reasoning)
- ◯ I don't know
- ◯ Other: _____

17. **Were new functionalities implemented during the migration?** *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ I don't know

18. **How many services did you plan to have on the new system?** *

_____

19. **What were the main challenges faced while DESIGNING the new system? Select up to 4 reasons that you consider most important** *

Tick all that apply.

- ☐ Identification of the boundaries of each service
- ☐ Decomposition of the existing system
- ☐ Automation support for testing
- ☐ Reduce coupling among services in the new architecture
- ☐ Finding the best Business-IT alignment
- ☐ Decomposition of the domain
- ☐ Finding the proper service granularity
- ☐ Lack of proper documentation of the existing system
- ☐ Bring domain experts into the process of designing the new system
- ☐ Uncommented code
- ☐ Database migration and splitting
- ☐ Communication among services
- ☐ Effort estimation
- ☐ DevOps infrastructure requires effort
- ☐ Library conversion effort
- ☐ Other: _____

20. **Please provide any additional information you find useful regarding the design of the new architecture**

_____

_____

_____

_____

_____

Implementing the new system

The objective of this section is to understand the process used to implement the new system

21. **How did you start the implementation?** *

Mark only one oval.

- ◯ Implementing new functionalities as microservices
- ◯ Re-implementing existing functionalities as microservices
- ◯ Re-implementing existing functionalities as minimum viable products
- ◯ Implementing the new system
- ◯ I don't know/I was not involved
- ◯ Other: _____

22. **How did you identify the first functionalities to migrate?** *

Mark only one oval.

- ◯ Less dependencies
- ◯ Less used by the users
- ◯ I don't know/I was not involved
- ◯ Other: _____

23. **What was the main process used to adopt the new system? If you used a mix of the below processes, please specify using "Other" option.** *

Mark only one oval.

- ◯ Phased adoption (having some functionalities on the new system and some on the old system)
- ◯ Parallel adoption (having all functionalities writing information on both systems at the same time and reading operations only from one, being able to easily switch between the systems)
- ◯ Big bang adoption (drop the old system and turn on the new one in a single step)
- ◯ I don't know/I was not involved
- ◯ Other: _____

24. **How did you consider the existing data in the new system?** *
Mark only one oval.

- ◯ The data was kept "as is" in the existing system
- ◯ The data was migrated to the new system by implementing data migration procedures. New services handle only new data
- ◯ The data was migrated to the new system by implementing data migration procedures. New services handle both old and new data
- ◯ I don't know/I was not involved
- ◯ Other: _____

25. **How many services did your final system have?**

_____

26. **What were the main challenges faced when IMPLEMENTING the new system? Select up to 4 reasons that you consider most important** *
Tick all that apply.

- ☐ Setting up the initial infrastructure for microservices to work
- ☐ Different thinking for developers
- ☐ Distributed monitoring
- ☐ Knowledge sharing, effective communication
- ☐ Distributed logging
- ☐ Distributed debugging
- ☐ Create uniformity across services
- ☐ Testing the new system
- ☐ Get the initial team to work together
- ☐ Using standards and norms
- ☐ Get the initial prototype working
- ☐ Other: _____

27. **Please provide any additional information you find useful regarding the implementation of the new system**

_____

_____

_____

_____

_____

Additional feedback

This section is completely optional

28. **Please provide any additional information about your experience while migrating to a microservices architecture**

29. **Please provide some feedback regarding the questionnaire**

30. **I want to subscribe this study results, and my e-mail is:**

Appendix B

Technical Challenges of Microservices Migrations - Research Validation

In the scope of a Master's Thesis on Software Engineering, research was conducted in order to identify the most common activities and challenges faced when adopting microservices.

You were identified as highly experienced in the field, and your opinion is essential and highly valued to validate the results of this study.

It should not take you more than 15 minutes and it will be crucial for the success of this study.

Results summary can be found in the following URL: https://bit.ly/2mEnDmx

Thank you!

*Required

1. **How many years of experience do you have?** *

2. **What is your current Job Title/Position?** *
   (Software Architect, Principal Engineer, Lead Engineer, Senior Engineer, etc.)

Main challenges evaluation

Please evaluate from 1 (strongly disapprove) to 5 (strongly approve) the main challenges identified. Please take into consideration the following description of value intervals:

[1- 2] The identified issues have no relation to microservices architecture or migration processes. The study does not bring value to the field.

[2-3] Some of the issues are related to the microservices architecture or migration processes, but there are essential issues missing. The study does not bring value to the field.

[3-4] The list of problems is complete and clear. The issues are related to the microservices architecture or migration processes, but some are not currently relevant. The study brings value to the field.

[4-5] The study identified the most common challenges of microservices architecture and migration. The study brings value to the area.

3. **Main challenges identified evaluation scale** *
   Please use a scale from 1 to 5. You can use non integer values.

4. **Please provide any additional feedback regarding the main challenges identified.**

Main solutions and best practices evaluation

Please evaluate from 1 (strongly disapprove) to 5 (strongly approve) the main solutions and best practices identified.
Please take into consideration the following description of value intervals:

[1-2] The solutions and best practices identified have no relation to microservices architecture or migration processes. The study does not bring value to the field.

[2-3] Some of the solutions and best practices identified are related to the microservices architecture or migration processes, but there are important techniques missing. The study does not bring value to the field.

[3-4]The list of solutions and best practices is complete and clear. The methods are related to the microservices architecture or migration processes, but some are not currently relevant. The study brings value to the field.

[4-5] The study identified the currently mostly used and proper techniques of microservices architecture and migration. The study brings value to the field.

5. **Main solutions and best practices identified evaluation scale** *
   Please use a scale from 1 to 5. You can use non integer values.

6. **Please provide any additional feedback regarding the main solutions and best practices identified.**

Distributed Transactions solution

Regarding one of the issues (Distributed Transactions management) a solution was implemented and proposed. It can be found in https://github.com/joaodiasneves/sapher/ with documentation in the readme file and github wiki pages.

Please provide your feedback.

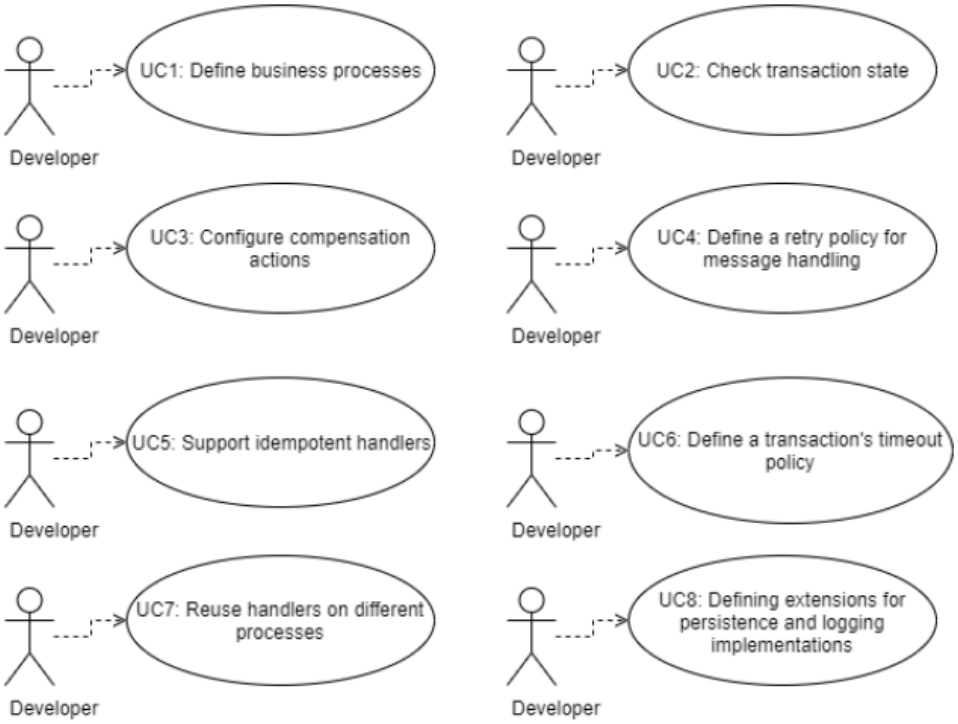
Non-functional requirements

1 - Provide reduced effort in adapting the solution to different implementation scenarios.
2 - The microservices architecture patterns and guidelines must be respected.
3 - Usage of choreographed sagas.
4 - The solution must be agnostic to communication channels.
5 - The solution must provide failure-handling mechanisms.

7. **How well did the presented solution achieve the non-functional requirements?** *

   Please use a scale from 1 to 5. You can use non integer values. (1 - Did not achieve them. 5 - Achieved them all).

Functional requirements



8. **How well did the presented solution achieve the functional requirements?** *

   Please use a scale from 1 to 5. You can use non integer values. (1 - Did not achieve them. 5 - Achieved them all).

9. **Please provide any additional feedback regarding the solution**

10. **Please provide any feedback regarding this questionnaire**

Appendix C

Distributed transactions solution implementation

One or more interested companies would use the defined solution in an attempt to mitigate the issues reported and categorised as data inconsistency or operations that were not entirely successful due to failed requests in the middle of a distributed transaction. From now on, this category of issues will be mentioned as sample issues (SI). SI can be raised by the final user's requests or by the system monitoring technology.

The mean of reported SI over one month of software usage would be recorded before and after the experiment, providing two distinct samples that can then be compared to evaluate the success of the delivered solution.

Preparation

The solution provided has the main objective of reducing the mean of reported SI.

In order to compare the mean of reported SI before and after the implementation of the solution, a paired t-test would be used. This kind of tests are utilised to compare two population means with two samples that can be matched with each other, which is usually the case with before and after measures of the same metric (Shier, 2004).

A paired t-test is a type of student's t-test which is used to determine the existence of a significant difference between two means, which is measured by the conventional statistic called Student's t, the larger the t the more significant the difference between sample means (Shier, 2004).

To apply a paired t-test, the following values are necessary:

$$n: \ number\ of\ samples\ from\ each\ population.$$

In this experiment, n will be the number of days during which the number of SI was observed and recorded. This should be at least 30.

Then, the mean difference between the two samples must be calculated. This is given by the following formula in which x is an element of each sample.

$$\bar{d} = \overline{\mu_1} - \overline{\mu_2} = \frac{\sum(x_1 - x_2)}{n}$$

The standard deviation s of the sample is also needed and can be obtained from the following formula.

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{d})^2}{n-1}}$$

On a paired t-test, to determine t, the following formula must be used. This is a composed formula as the lower part of the division ($\frac{s}{\sqrt{n}}$) is the calculation of the standard error of the means difference (Shier, 2004).

$$t = \frac{\bar{d}}{\frac{s}{\sqrt{n}}}$$

The resulting t value is then matched against a t table according to the desired significance level value and if the test is one-tailed or two-tailed (Shier, 2004).

Considering that μ_a is the mean of reported SI during one month after the solution was implemented, and μ_b the mean of reported SI during one month before the solution was implemented, the following hypotheses can be reached.

$$H_0: The\ average\ number\ of\ reported\ SI\ remained\ the\ same\ after\ the\ solution\ was\ implemented$$

$$H_0: \mu_a - \mu_b = 0$$

$$H_1: The\ average\ number\ of\ reported\ SI\ was\ reduced\ after\ the\ solution\ was\ implemented$$

$$H_1: \mu_a - \mu_b < 0$$

If the mean of reported SI is reduced, then $H_0$is refuted, and therefore it is valid to say that the solution brings value to the user by reducing the number of reported SI.